The TEXbook

DONALD E. KNUTH

Illustrations by
DUANE BIBBY



ADDISON–WESLEY
PUBLISHING COMPANY

Reading, Massachusetts
Menlo Park, California
New York
Don Mills, Ontario
Wokingham, England
Amsterdam · Bonn
Sydney · Singapore · Tokyo
Madrid · San Juan

VISIT…

LANZAROTE
Caliente.com

This manual describes TeX Version 3.0. Some of the advanced features mentioned here are absent from earlier versions.

The quotation on page 61 is copyright © 1970 by Sesame Street, Inc., and used by permission of the Children's Television Workshop.

TeX is a trademark of the American Mathematical Society.

METAFONT is a trademark of Addison–Wesley Publishing Company.

Library of Congress cataloging in publication data

```
Knuth, Donald Ervin, 1938-
   The TeXbook.

   (Computers & Typesetting ; A)
   Includes index.
   1. TeX (Computer system).  2. Computerized
typesetting.  3. Mathematics printing.  I. Title.
II. Series:  Knuth, Donald Ervin, 1938-    .
Computers & typesetting ; A.
Z253.4.T47K58  1986         686.2'2544      85-30845
ISBN 0-201-13447-0
ISBN 0-201-13448-9 (soft)
```

Twentieth printing, revised, May 1991

Copyright © 1984, 1986 by the American Mathematical Society

This book is published jointly by the American Mathematical Society and Addison–Wesley Publishing Company. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publishers. Printed in the United States of America.

ISBN 0-201-13448-9
TUVWXYZ–DO–97654321

前言

尊敬的读者:这是关于 TEX 的工具书。 TEX 是为了更好地排版书籍——特别是包含很多数学公式的书籍而设计的一个软件。用 TEX 的格式准备好书稿后, 你就准确地告诉了计算机怎样把它变成书页, 而得到的排版质量可以与世界上最好的排版工人的效果相媲美; 但是你所做的工作仅仅相当于在普通打字机上把书稿直接输入进去。实际上, 如果你算上通常修改打印稿所花费的时间, 那么你的工作量可能非常小, 这是因为计算机的文本文件修改并重新排版是相当容易的。(如果这样的好话听起来不象真的, 那么记住这是 TEX 设计者所声明的, 如果有一天凑巧要用到 TEX 时, 这些承诺可能有点偏差; 但是总之要看下去。)

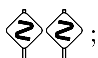
这个手册是为以前从未使用过 TEX 以及有经验的 TEX 高手所使用的。换句话说, 它是想当人人需要的万金油, 当然也有可能是无用的垃圾。你需要知道的 TEX 的每个细节都写在本书的某个地方, 并且因此许多东西是大多数用户所不关心的。如果你正在写一个简单的草稿, 那么就根本不需要知道太多的 TEX; 另一方面, 某些排版专业书籍的技术本来就很难, 而且如果你想得到更复杂的效果, 就应该洞悉 TEX 的某些鲜为人知的知识。为了使不同的用户能高效地阅览本手册, 我们用一个特殊的标志来标记那些只面向高手的内容: 当在某段的开头出现符号



时, 它就是路边上“转弯危险”的警告; 如果你不需要, 就别去看它。在驾驭 TEX 方面勇敢且有经验的高手可逐渐地进入这些危险的区域, 但是对大多数使用者而言, 这些细节是无关紧要的。

在开始阅读之前, 所有你真正应该知道的是怎样在计算机上用标准的文本编辑程序得到一个文本文件。这个手册说明了什么样的文件 TEX 才能识别, 但是计算机的基本用法这里没有。如果你打算用 TEX 完成繁重的数学工作, 前面提到的一些打字技术是非常有帮助的, 但是也不是绝对必需的。TEX 将自动完成大部分必要的公式编排; 但是经验丰富的用户可得到更好的效果, 因为可以用许多种方法来编排公式。

本手册中, 有些段落深奥到了

;

对它们, 单个危险标记的所有忠告都要加倍。在你试图进入系统的如此双重危险的部分前, 大概应该至少使用一个月的 TEX; 实际上, 大多数用户从不需要这么详细地了解 TEX——即使他们每天都用。毕竟, 不知道发动机怎样工作也可以开汽车。但是, 如果你爱钻研, 那么所有东西都在这里。(当然是关于 TEX 的, 而不是关于汽车的。)

设置这样不同的复杂等级的原因在于随着人们不断熟悉这个强大的工具, 他们自己也在改变。当你首次使用 TEX 时, 会发现它的某些东西非常简单, 而其它的东西就得花些功夫。一天以后, 当你能排版几页之后, 你就进了一步; 以前困扰你的概念现在看起来就是简单自然的, 并

且你能在机器得出结果前在脑海中勾画出最后的结果。但是你可能陷入另一种挑战。再过一个星期, 你的看法就又变了, 而且进入另外一个方面; 等等。数年过去后, 你可能掌握了许多不同的排版方式中; 并且发现随着经验的丰富, TEX 的用法也在不断变换。这就是工具之所以强大的原因: 总有更多的东西需要学习, 并且总有更好的方法来处理你以前的工作。在每个发展阶段, 你都希望有一个稍微不同的手册, 可能甚至希望亲自写一本。只要留意本书的危险标记, 你就能在特定的阶段更好地把精力集中在感兴趣的地方。

计算机系统手册一般读起来比较乏味, 但是振作起来: 本手册偶尔给你几个笑话, 所以你实际上可能喜欢读一读。(但是, 只有你理解了所用的技术要领, 大多数笑话才能真正被欣赏——所以要仔细点啊。)

本手册的另一个值得注意的特征是, 它并不总是那么确切。当非正式地引入 TEX 的某些概念时, 我们给出大致的规则; 在后面, 你会发现这些规则并不严格正确。一般地, 后面的章节比前面的章节所描述的更确切。作者感到, 这种善意的谎言实际上使你更容易接受这些概念。一旦你吃透了一个明显错误的规则, 那么不难把那个规则的例外情形补充上。

为了帮助你吸收所学的知识, 大量的练习都散布在整个手册中。一般希望每个读者把每个练习都做一下, 当然除了出现在"危险"部分的问题。如果你做不出哪个问题, 那么可以去查阅答案。但是务必请先自己做一下; 那么你会更快地学会更多的知识。还有, 如果你认为的确得到了答案, 那么应当到附录 A 去对一下答案, 仅仅是确认一下。

在本书中所用的 TEX 代码类似于作者第一次在文档格式代码的尝试, 但是新系统与旧系统在差不多几千个细节上有差别。两种代码都叫做 TEX; 但是以后旧代码应该称为 TEX78, 并且将很快淡出。我们把名字 TEX 保留给这里陈述的代码, 因为它是如此优秀并且不再改动。

我要感谢帮助我完成这个 TEX 代码"成熟版"的几百个人, 因为本版本是以他们的经验以及系统的初版为基础的。我在 Stanford 的工作得到了 National Science Foundation, Office of Naval Research, IBM Corporation 和 System Development Foundation 的大力支持。我还要感谢 American Mathematical Society 的鼓励, 并谢谢它建立 TEX Users Group 以及出版 TUGboat 快报(见附录 J)。

Stanford, California — D. E. K.
June 1983

目录

1 此名有诗意 1
2 书籍排版与普通排版 2
3 TEX 的控制系列 5
4 字体类型 9
5 编组 14
6 运行 TEX 17
7 TEX 工作原理 28
8 字符输入 33
9 TEX 的 Roman 字体 38
10 尺寸 42
11 盒子 46
12 粘连 50
13 模式 63
14 分段为行 68
15 组行为页 83
16 数学公式 98
17 数学排版进阶 107
18 精致的数学排版 126
19 列表公式 146
20 定义(宏) 158
21 盒子制作 178
22 对齐方式 186
23 输出例行程序 205

24 垂直模式汇总 218
25 水平模式汇总 235
26 数学模式汇总 238
27 错误修复 243

附录

A 练习答案 .... ?
B 基本控制系列 .... ?
C 字符代码 .... ?
D 诡计多端 .... ?
E 版式举例 .... ?
F 字体表 .... ?
G 公式的盒子 .... ?
H 连字符 .... ?
I 索引 .... ?
J 加入 TEX 社团 .... ?

1. 此名有诗意

英语单词“technology”来源于以字母 $\tau\epsilon\chi\ldots$ 开头的希腊词根; 并且这个希腊单词除了 technology 的意思外也有 art 的意思。因此, 名称 TEX 是 $\tau\epsilon\chi$ 的大写格式。

在发音时, TEX 的 χ 的发音与希腊的 chi 一样, 而不是“x”, 所以 TEX 与 blecchhh 押韵。“ch” 听起来象苏格兰单词中的 *loch* 或者德语单词中的 *ach*; 它在西班牙语中是“j”, 在俄语中是“kh”。当你对着计算机正确读出时, 终端屏幕上可能有点雾。

这个发音练习是提醒你, TEX 主要处理的是高质量的专业书稿: 它的重点在艺术和专业方面, 就象希腊单词的含义一样。如果你仅仅想得到一个过得去——可读下去但不那么漂亮——的文书, 那么简单的系统一般就够用了。使用 TEX 的目的是得到最好的质量; 这就要在细节上花功夫, 但是你不会认为它难到哪里去, 并且你会为所完成的作品感到特别骄傲。

另一方面重要的是要注意到与 TEX 名称有关的另一件事: “E”是错位的。这个偏移“E”的标识提醒人们, TEX 与排版有关, 并且把 TEX 从其它系统的名称区别开来。实际上, TEX (读音为 *tecks*) 是 Honeywell Information Systems 的极好的*Text EXecutive*处理器。因为这两个系统的名称读音差别很大, 所以它们的拼写也不同。在计算机中表明 TEX 文件的正确方法, 或者当所用的方式无法降低“E”时, 就要写作“TeX”。这样, 就与类似的名称不会产生混淆, 并且为人们可以正确发音提供了条件。

▸ **练习1.1**
当你已经掌握本书中的知识后, 你属于 TEX 专家还是 TEX 爱好者?

2. 书籍排版与普通排版

当你第一次使用计算机终端时, 可能不得不校准数字“1”和小写“l”的差别。当你进行到一般书籍出版的排印步骤时, 还要做出几个同样的校准; 你的眼睛和手指需要知道这几个差别。

首先, 在书中有两种引号, 但是在打字机上只有一种。即使在比普通打字机更多的字符的计算机键盘上, 也可能只有一种无方向的双引号 ("), 因为计算机的标准 ASCII 码不是专门为书籍出版所发明的。但是, 你的计算机可能确实有两种不同的单引号, 即 ‘ 和 ’; 第二个也用在省略语中。美国键盘包含一个看起来象 ` 的左引号和一个看起来象 ' 或 ´ 的省略符或右引号。

为了用 TeX 得到双引号, 可直接键入两个相应类型的单引号。比如, 为了得到习惯用语

“I understand.”

(包括双引号), 应该在计算机中键入

```
``I understand.''
```

类似打字机风格将在本手册通篇使用, 以显示可在你的终端上键入的 TeX 的用词, 使得实际键入的符号容易从 TeX 产生的输出和手册自身的注解区分开。下面是在例子中要用到的符号:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789"#$%&@*+-=,.:;?!
()<>[]{}`'\|/_^~
```

如果你的计算机终端凑巧不能包含所有这些字符, 别失望; TeX 可设法应付它。另外一个符号

␣

用来表示一个 *blank space* (空格)——如果要重点强调所键入的空格; 因此, 你真正要在上面的例子中键入的是

```
``I␣understand.''
```

如果没有这样的符号, 就很难知道某些用词的看不见的部分。但是我们不会太频繁使用“␣”, 因为空格一般足以被看见。

书籍排版和普通排版在虚线, 连字符和减号方面明显不同。在优秀的数学书籍中, 这些符号都是不同的; 实际上, 一般至少有四种不同的符号

连字符 (-);
短破折号(–);
破折号 (—);

减号 (−).

连字符用在象“daughter-in-law”和“X-rated”这样的复合单词中。短破折号用在象“pages 13–34”以及象本书中“exercise 1.2.6–52”这样的数字区间方面。破折号用作句子中的标点——就是我们通常所称的简单破折号。而减号用在公式中。原则性强的 TEX 用户要仔细区分这四种用法, 说明如下:

对连字符, 键入一个连字符 (`-`);
对短破折号, 键入两个连字符 (`--`);
对破折号, 键入三个连字符 (`---`);
对减号, 把连字符放在数学模式中 (`$-$`).

(数学模式出现在 dollar 符号之间; 这将在后面讨论, 因此你现在不必考虑。)

▸ **练习2.1**

看看怎样在 TEX 中键入下列句子: Alice said, “I always use an en-dash instead of a hyphen when specifying page numbers like ‘480–491’ in a bibliography.”

▸ **练习2.2**

当你在一行中连续键入四个连字符时, 会得到什么结果?

如果你仔细查看大多数排版精良的书籍, 就会发现某些字母组合会当成一个字母来处理。例如, “find”中的“f”和“i”就是这样。这样的组合称为组合字 (ligature), 并且专业排版工已经培训得习惯于诸如 `ff`, `fi`, `fl`, `ffi`, 和 `ffl` 的字母组合。(原因是象“find”这样的单词在大多数排版风格上不太好看,而用组合字代替这些字母就不存在那样的问题了。有点令人惊讶的是, 习惯的组合字经常出现在英语中; 在其它语言中, 别的组合也很重要。

▸ **练习2.3**

给出一个包含两个组合字的英语单词。

还好, 你不必管这些组合字: TEX 完全有能力自己解决这些问题, 方法同把“`--`”转换成“–”一样。实际上, 为了得到更好的观感, TEX 还把应当更接近的相邻字母(象紧跟“`V`”的“`A`”)看作组合; 这称为字距调整。

总结: 当用 TEX 直接排版时, 就象在普通打字机上键入书稿一样, 但是需要注意引号, 数字 1 和各种连字符/破折号。象组合字和字距调整这些其它的精细调整由 TEX 自动完成。



(你准备好看本段了吗? 这里的“危险”标志就是提醒你, 这些材料应当在首次阅读时跳过去。可能已经是第二遍了吧。提醒读者的这些段落有时用到了后面章节中的概念。)

如果你的键盘没有左引号符号, 那么你可以键入 `\lq`, 如果下一个字符是字母, 就应该紧接一个空格, 如果下一个字符是空格, 就应该紧接一个 `\`。类似地, `\rq` 得到一个右引号字符。清楚了吗?

```
\lq\lq␣I␣understand.\rq\rq\␣
```

如果你想键入引号中的引号, 比如右单引号后跟一个右双引号, 那么你不能直接键入 `'''`, 因为 TEX 将把它看成 ”’(即右双引号后面跟右单引号)。如果你已经看过第5章, 可能认为解决之道是采用编组——即键入所谓的 `{'}''`。但是, 却得不到想看到的结果, 因为单右引号后面的间距比双右引号后面的要小: 得到的是 ’”, 这的确是一个单引号后面跟一个双引号(如果你仔细观察的话), 但是看起来却几乎是三个一样的单引号。另一方面, 当然也不要键入 `'␣''`, 否则间距太大——就象单词之间的空格那么大, 并且当 TEX 在生成段落时会在这样的空格处断行! 解决的方法是键入 `'\thinspace''`, 就得到所要的 ’ ” 了。

▶ 练习2.4

好了, 现在你知道怎样得到 ”’ 和 ’ ” 了; 那么你怎样得到 “‘ 和 ‘“ 呢?

▶ 练习2.5

想一下作者为什么用控制系列`\thinspace`来解决相邻引号问题而不是用更 巧妙的 `'$\,$''` (它也可以胜任)?

3. TEX 的控制系列

你的键盘所含的字符与所期望得到的符号相比太少了。为了充分利用有限的键盘, 你可以键入的某个字符被保留为特殊用途, 即所谓的转义符(escape character)。只要你要键入控制文稿格式的信息或不能直接使用键盘的内容, 就应该键入转义符, 再在后面跟上对应的指令。

注意: 有些计算机终端上有一个标记为'ESC'的键, 但不是你的转义符! 它是一个向操作系统发送特殊指令的键, 所以不要把它与本手册的"转义(escape)"混淆。

TEX 允许使用任何字符作为转义符, 但是一般采用反斜线字符'\', 因为反斜线较方便键入且在一般文本中很少使用。当不同的 TEX 用户统一规则时, 做事就最有效, 所以我们将在本手册的所有例子中使用反斜线字符作为转义符。

键入'\'后, 紧接(即在转义符后面紧接)着键入一个代码命令来告诉 TEX 你的想法。这样的命令称为*控制系列*。例如, 你可以键入

```
\input MS
```

(就象我们后面将讲述的那样)它让 TEX 开始读入名字为'MS.tex'的文件; 字符串 '\input'是一个控制系列。另一个例子是

```
George P\'olya and Gabor Szeg\"o.
```

TEX 把它转换为'George Pólya and Gabor Szegö.'。 这里有两个控制系列, \' 和 \"; 这些控制系列用来把重音号放在字母上。

控制系列有两种类型。第一类象\input一样, 称为*控制词*; 它由转义符后跟一个或多个*字母*组成, 后面跟一个空格或非字母的字符。(TEX 必须知道控制系列何时结束, 因此, 如果下一个字符是字母, 你必须在控制词后加一空格。例如, 如果你键入'\inputMS', TEX 将认为它是一个7个字母组成的控制词。) 要是对"字母"有疑问, 答案就是, TEX 一般把52个字母 A...Z 和 a...z 看作字母。数字 0...9 不被看作字母, 所以不能出现在第一类控制系列中。

另一类控制系列, 象 \' 一样, 被称为*控制符号*; 它由转义符后跟一个单个*非字母*组成。在这种情况下, 你不必用空格把控制系列与其后的字母分开, 因为第二类控制系列在转义符后永远只有一个符号。

▸ **练习3.1**

在'\I'm \exercise3.1\\!'中, 控制系列有哪些?

▸ **练习3.2**

我们已经知道键入 P\'olya 就得到'Pólya'。你猜出来法语单词'mathématique' 和'centimètre'怎样键入吗?

当空格出现在控制词(一个完全由字母组成的控制系列)后面时, 它被 TEX 所忽略; 即, 它不会被看作属于所排版文稿的“实际”空格。但是, 当空格出现在控制符号后面时, 它就是一个实际空格。

现在, 出现问题了, 如果真要在控制词后面得到一个空格该怎样做呢? 后面我们会看到, TEX 把两个或多个连续空格看作单个空格, 所以“键入两个空格”不解决问题。 正确的方法是键入“控制空格”, 即

```
\␣
```

(转义符后面跟一个空格); TEX 将把它看作不能忽略的空格。注意, 控制空格是第二类控制系列, 即控制符号, 因为跟在转义符后面的是一个单个非字母(␣)。两个连续空格被看作等价于单个空格, 所以跟在 \␣ 后的多余的空格将被忽略。但是, 如果你要在文稿中键入3个连续空格, 就应该键入‘\␣\␣\␣’。顺便说一下, 打字员通常在句子后面要键入两个空格; 但是以后我们将知道, 在这种情况下, TEX 用自己的方式来得到额外的间距。 因此, 你不需在意所键入的空格数目的一致性。

象 ⟨return⟩ 这样不能显示出来的控制符也可以跟在转义符后面, 并且按照上面的规则, 它们是一些独特的控制系列。TEX 从开始就把 \⟨return⟩ 和 \⟨tab⟩ 看作同 \␣ (控制空格)一样; 这些特殊的控制系列不可能重新被定义, 因为在文件中无法区分它们。

对你而言, 通常不需要使用“控制空格”, 因为控制系列不经常出现在单词的尾部。但是, 这里有一个这方面的典型例子: 本手册本身是用 TEX 排版的, 并且相当频繁出现的一个东西就是乖巧的‘TEX’, 它要求回退间距和降低 E。这是一个特殊的控制词

```
\TeX
```

为了排版‘TEX’, 它使用了大概半打的指令。当要排版出‘TEX ignores spaces after control words.’这样的语句时, 所键入的文稿如下:

```
\TeX\ ignores spaces after control words.
```

注意, TEX 后面有一个额外的 \; 因为 TEX 忽略掉了控制词后面的空格, 所以它生成了必需的控制空格。如果没有这个额外的 \, 结果就是

TEXignores spaces after control words.

另外, 不能在任何情况下都在 TEX 后面直接跟一个 \。比如, 考虑一下下列语句

```
the logo ‘\TeX’.
```

在这种情况下, 额外的 \ 根本就不对; 实际上, 如果你键入

```
the logo ‘\TeX\’.
```

就会得到稀奇古怪的结果。 你知道得到什么吗? 答案是: \' 是重音符的控制系列, 就象上面例子中的 P\'olya 一样; 因此结果是在下一个非空白的字符——凑巧是句点——上面加重音。换句话说, 你得到一个带重音的句点, 排版结果是

the logo 'TEX.́

计算机正确地执行了后面的指令, 但没有正确表达你的意思。

TEX 能执行大约 900 个控制系列, 这是它的内置语汇的一部分, 所有这些控制系列都阐述在本手册的某个地方。但是别为要掌握这么多不同的内容而担心, 因为只要不是对付不常用的复杂排版, 你是不需要掌握很多控制系列的。再者, 你需要实际掌握的那些只在相对少的几个种类中, 所以它们不太难理解。例如, 许多控制系列就是数学公式中特殊字符的名字; 键入'\pi'就得到 π, '\Pi' 得到 'Π', '\aleph' 得到 '$\aleph$', '\infty' 得到 '∞', '\le' 得到 '$\le$', '\ge' 得到 '$\ge$', '\ne' 得到 '$\ne$', '\oplus' 得到 '$\oplus$', '\otimes' 得到 '$\otimes$'。附录 F 包含了这类符号的几个列表。



在控制系列的名称中, 大写和小写字母之间没有内在的等价关系。例如, '\pi', '\Pi, '\PI' 和 '\pI' 是四个不同的控制词。

刚刚提到的大约 900 个控制系列并不是全部, 因为可以容易地定义更多的控制系列。例如, 如果你想用自己喜欢的名字来定义数学符号, 那么就更好记住它们, 你可以随心所欲地命名它们; 第 20 章告诉你这方面的内容。

大约 300 个控制系列称为*原始控制系列*; 它们是不能分解为更简单作用的低级控制系列。所有其它控制系列归根结底都由这些原始控制系列来定义。例如, \input 是原始控制系列, \' 和 \" 却不是; 后者用原始控制系列 \accent 来定义。

在文稿中, 人们几乎用不到 TEX 的原始控制系列, 因为原始控制系列是那么 . . . 地*原始*。如果你试图做 TEX 的低级事情, 就必需键入许多指令; 这浪费时间且容易犯错误。一般最好利用规定相应功能的高级控制系列, 而不要每次都把得到本功能的所有东西都键入。高级控制系列只需要用原始控制系列定义一次。例如, \TeX 是"排版 TEX 的标识符"的控制系列; \' 是"在下一字符上放重音"的控制系列; 并且当排版格式变化时, 所有这些控制系列只需原始控制系列的不同组合即可。如果 TEX 的标识符要改变, 作者只必须改变一个定义, 并且改变后的东西自动出现在需要出现的地方。相反, 如果它每次作为一系列的原始控制系列而确定时, 改变标识符将要花费大量劳力。

在更高级方面, 有管理文档整个格式的控制系列。例如, 在前面作者在给出每个练习之前, 键入了'\exercise';

- 算出练习的编号(比如, 对第三章的第二个练习得到'3.2');
- 在同一行上, 用相应的字体排版出'▸ **EXERCISE 3.2**', 并且让三角形顶左边;
- 在行前留出额外小间距, 如果需要的话在此行换页;

- 禁止在此行后换页;
- 保证接下来的行不缩进。

明显的好处是不必每次键入所有这些单个指令。并且因为本手册完全用高级控制系列所描述, 所以通过改变大约一打的定义, 就可以用完全不同的格式来打印出它来。

怎样从高级控制系列中区分出 TEX 的原始控制系列呢? 有两种方法: (1) 本手册的索引罗列了讨论过的所有控制系列, 每个原始控制系列都用星号标记了。 (2) 当运行 TEX 时, 你可以显示出某个控制系列的含义。如果你键入'\show\cs'(其中 \cs 是任意控制系列), TEX 将回答出它的当前含义。例如, '\show\input'得到'> \input=\input.', 这是因为 \input 是原始控制系列。另一方面, 'show\thinspace'得到

```
> \thinspace=macro:
->\kern .16667em .
```

这意味着 \thinspace 已经被定义为'\kern .16667em '。通过键入 '\show\kern', 你可以验证 \kern 是原始的。\show的结果出现在你的终端和运行 TEX 结束后的 log 文件中。

▶ **练习3.3**
控制系列 \␣ 和\⟨return⟩中, 哪个是原始的?

在后面的章节, 我们将频繁讨论"plain TEX"格式, 它大约有 600 个基本控制系列, 这些都在附录 B 中定义出。 当 TEX 开始处理文稿时, 这些控制系列及 300 个左右的原始控制系列通常一起出现, 这就是为什么 TEX 在一开始声称大概有 900 个控制系列。我们将看到 plain TEX 怎样被用来得到灵活格式的文档以满足不同人们的需要, 不过要使用 TEX 系统所带的一些字体。但是要记住, plain TEX 只不过是在 TEX 的原始控制系列上设计的无数格式的一种; 如果想要其它格式, 通常可以改编 TEX 以得到你心中想要的。领会的最好方法可能是从 plain TEX 开始, 随着经验的积累, 一点一点地修改它的定义。

附录 E 中含有格式的例子, 它们可以与附录 B 一起满足特殊要求; 例如, 其中有一组适用于商业通信的定义。排版本书稿的完整的格式规定也在附录 E 中。因此, 如果你的目的是设计 TEX 格式, 在学会附录 B 后你可能要研究研究附录 E。在经过定义控制系列的知识的训练后, 你就可以为其他人们编写一些格式; 那么你应该为本手册写一个附录, 解释一下你的设计规则。

就 TEX 新手所关心的而言, 这些讨论的重点是, 的确可以定义一些非标准的 TEX 控制系列。当 TEX 讲什么属于"plain TEX"时, 就意味着 TEX 不是只能按照这种方法才可; 你可以通过改变附录 B 中一个或多个定义来改变结果。但是在你成为有经验的 TEX 排版专家前, 确实可以依赖 plain TEX 的控制系列。

▶ **练习3.4**
有多少 2 个字符的不同控制系列(转义符算一个)? 有多少 3 个字符的?

4. 字体

有时候, 你想把字体从一种变成另一种, 例如当你要 **bold** 或 *emphasize* 某些内容时。TEX 能处理 256 个字符的集合——所谓的字体, 并且用控制系列来选择特殊的字体。例如, 你可以用下列方法来指定几个词的字体(采用附录 B 的 plain TEX 格式):

```
to be \bf bold \rm or to \sl emphasize \rm something.
```

to be **bold** or to *emphasize* something.

Plain TEX 提供了下列控制系列来改变字体:

`\rm` 转变为标准"roman"字体:	Roman
`\sl` 转变为斜 roman 字体:	*Slanted*
`\it` 转变为 italic 字体:	*Italic*
`\tt` 转变为类似 typewriter 的字体:	`Typewriter`
`\bf` 转变为 bold 的字体:	**Bold**

在开始排版前, 如果你不定义为其它字体, 将使用 roman 字体 (`\rm`)。

注意, 为了起强调作用, 这些类型中的 roman 和 *Slanted* 字体之间有一个倾斜, 而它们本质上是同一个字体, 仅仅字母有一些歪斜; 而 *italic* 字体与它们在本质上不是一个字体。(通过观察 unslanted italic 字体的字母, 你可能会更好地体会到 roman 和 italic 字体之间的差别。) 目前排版业行规处于变革时代, 因为新技术把过去不可能的工作变得可能; 人们正致力于解决使用新排版技术的成本问题。Slanted roman 字体出现在 1930 年代, 但是直到 1970 年代后期, 它才作为传统 italic 字体的替代品而被广泛使用。在数学文稿中使用它是有益的, 因为 slanted 字母可以与数学公式中的 italic 字母区别开。在两个不同目的下使用同一 italic 字体——例如, 定理的陈述和定理中变量的名称都使用 italic——将产生冲突, 现在可以用 slanted 类型字体来避免了。人们一般对 slanted 相对于 italic 字体的用处不怎么认可, 但是 slanted 类型正成为参考文献中的书名和杂志名的受欢迎的字体。

特殊字体有利于强调, 但是不利于阅读; 如果本手册的很长一部分完全设置为 bold 或 slanted 或 italic 字体, 你的眼睛会很累。因此, roman 字体占据了大部分排版内容。但是, 当要回到 roman 字体时每次都键入'`\rm`'太费事, 于是 TEX 给出了一个简单的办法, 利用"大括号"符号: 你可以在特殊符号 `{` 和 `}` 中转换字体而不影响括号外的字体。例如, 表达本章上面的语句一般用

```
to be {\bf bold} or to {\sl emphasize} something.
```

这是一般编组思想的特殊例子, 我们将在下一章讨论它。最好把改变字体的前一种方法忘掉, 而使用编组; 这样, 你的 TEX 文稿看起来更舒服, 并且可能从不(或几乎从不)要键入'`\rm`'。

▶ **练习4.1**

看看怎样键入参考文献'Ulrich Dieter, *Journal für die reine und angewandte Mathematik* **201** (1959), 37–70.' [利用编组。]

在上面的讨论中, 我们已经阐述了排版质量的一个重要方面。例如, 观察本句中的 *italicized* and 和 *slanted* words 两组词。因为 italic 和 slanted 字体向右倾斜, 字母 d 刺进了它与随后 roman 字体的单词之间的空格; 因此, 间距看起来就不够了, 虽然字母都是正确的。为了补偿损失的间距, TEX 允许就在转换回 unslanted 字母前放一个控制系列'`\/`'。当你键入

```
{\it italicized\/} and {\sl slanted\/} words
```

时, 就得到 *italicized* and 和 *slanted* words, 更好看了。'`\/`'告诉 TEX 要按不同字母对前一字母作一个"*italic correction*"; 对标准的 italic 字体, 这个校正对 f 是对 c 的 4 倍。

有时候, 不想要 italic 校正, 因为其它的因素使得在视觉上不感松散。标准的上策是只在从 slanted 或 italic 转换到 roman 或 bold 前使用 `\/`, 除非下一个字符是句点或逗号。例如, 键入

```
{\it italics\/} for {\it emphasis}.
```

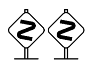
在以前的手册格式中, 单词后的标点与单词的字体相同; 但是 italic 的分号一般很难看, 所以这个约定被改过来了。当一个 italicized 单词正好出现在分号前时, 作者建议使用 '`{\it word\/};`'。

▶ **练习4.2**

看看怎样在 italicized 句子中间键入一个 roman 单词(*Explain how to typeset a* roman *word in the midst of an italicized sentence.*)。

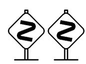


每种字体的每个字母都有 italic 校正, 你可通过键入 `\/` 来激活它。在 unslanted 字体中校正一般是零, 但是有一些例外: 为了排版引号中的 bold '**f**', 你应该键入 `bold '{\bf f\/}'`, 以免得到的是bold '**f**'。



▶ **练习4.3**

定义一个控制系列 `\ic` 使得'`\ic c`'把字符 c 的 italic 校正赋值给 TEX 的记数器 `\dimen0`。



原始控制系列 `\nullfont` 代表无字符的一种字体。如果你没有给定其它字体, 这种字体总会出现。

字体的外形随字体大小而变。例如, 你现在所看到的是所谓"10-point'字体, 因为在用打印机的单位测量时, 它的设计上的某些特征分别是 10 points。(后面我们将阐述 point system; 现在, 只需指出, 本句的括号的高度正好是 10 points—且单破折号的宽度正好是 10 points。) 本手册的"危险"标识的段落设置为 9-point 字体, 脚注为 8-point, 上标为 7-point 或 6-point, 上上标为 5-point。

在 TEX 文稿中使用的每个字体都伴随一个控制系列; 例如, 本段的 10-point 字体叫做 `\tenrm`, 而且相应的 9-point 字体叫做 `\ninerm`。与 `\tenrm` 和 `\ninerm` 对应的 slanted 字体叫做

tensl 和 ninesl。这些控制系列不是 TEX 内建的, 也不是它们的真实字体名; 当需要在文稿中引入新字体时, 只建议 TEX 用户使用便利的名字即可。这样的控制系列的作用是改变字体。

当同时使用不同大小的字体时, TEX 将按照它们的"基线"把字母排成一行。例如, 如果你键入

```
\tenrm smaller \ninerm and smaller
\eightrm and smaller \sevenrm and smaller
\sixrm and smaller \fiverm and smaller \tenrm
```

得到的是 smaller and smaller and smaller and smaller and smaller and smaller。当然, 这是作者和读者都不习惯的样子, 因为打印机不能象传统的铅字排版那样。可能希望用 a still small voice 诵读的诗人会为未来的诗作而采用频繁变幻的字体, 但是现在仅仅是 (like the author of this manual) 做实验来使用这样古怪的字体。要没有更好的理由, 你不应该因追求字体转换的效果而忘记主要任务。

小心的读者对这点可能会感到困惑, 因为在本章开头我们说'\rm'是转换到 roman 字体的命令, 但是后面我们又说'\tenrm'也是如此。事实上两种方法都可以。但在习惯设置上, \rm 的意思是'转换到当前大小的 roman 字体'; 而 \tenrm 的意思是"转换到 10-point 的 roman 字体"。在 plain TEX 格式中, 只提供了 10-point 字体, 所以 \rm 总是等价于 \tenrm; 但在更复杂的格式中, \rm 的意思在文稿的不同部分是不同的。例如, 在作者排版本手册所用的格式中, 有一个控制系列'\tenpoint', 它导致了 \rm 等于 \tenrm, \sl 等于 \tensl, 等等; 而'\ninepoint'也改变了上述定义使得 \rm 等于 \ninerm 等等。还有另外一个控制系列, 它用在每章的结尾的引用语上; 当引用语出现时, \rm 和 \sl 临时分别变成 8-point unslanted sans-serif type 和 *8-point slanted sans-serif type*。暗中常常重新定义简写 \rm 和 \sl 的这种方法使得排版者不需要记住现在需要使用的字体及其大小。

▶ **练习4.4**

你知道为什么作者选用'\tenpoint'和'\tenrm'等等这样的名字, 而不使用'\10point'和'\10rm'呢?



▶ **练习4.5**

假设你在一个文稿中用 slanted 字体表示强调, 但是编辑却突然告诉你把所有的 slanted 改成 italic。怎样用简单的方法来实现它?



每个字体都有一个外部名字以区别于这个特殊库中的所有其它字体。例如, 本句中的字体(英文版)称为'cmr9', 它是"Computer Modern Roman 9 point"的缩写。为了让 TEX 使用本字体, 在附录 E 中出现了下列命令

```
\font\ninerm=cmr9
```

一般地, 你用'\font\cs=⟨external font name⟩'把一个特殊字体的有关信息载入到内存中; 然后控制系列 \cs 将在排版时选择那个字体。Plain TEX 最初只定义了 16 个字体(见附录 B 和 E), 但是你可以用 \font

把你的系统字体库中的任何字体为 TEX 所用。

通过放大或缩小字符的图像, 通常一种字体可以有几种不同的大小。每种字体有一个所谓设计尺寸, 它表示其正常尺寸的缺省值; 例如, `cmr9` 的设计尺寸是 9 points。但是在许多系统中, 通过放大和缩小尺寸, 在一个大小范围内也可以使用某一特定字体。为了把缩放后的字体载入 TEX 的内存, 你只需直接使用'`\font\cs=`⟨external font name⟩ `at` ⟨desired size⟩'。例如, 命令

```
\font\magnifiedfiverm=cmr5 at 10pt
```

把两倍于正常尺寸的 5-point Computer Modern Roman 载入。(注意: 在使用这个'`at`'属性时, 你应该确保你的排字机支持所讨论的那个尺寸的字体; TEX 容许小于 2048 points 的正值的任意 ⟨desired size⟩, 但是如果你的打印设备不支持缩放的字体, 输出结果就不会正确。

在 `cmr5 at 10pt` 和正常的 10-point 字体 `cmr10` 之间有什么差别呢? 很多呀; 设计精良的字体在不同尺寸的图像是不同的, 并且为了提高观感, 字母通常有不同的相对高度和宽度。

Ten point type is different from magnified five-point type.

通常最好只相对于设计尺寸微微放大字体, 除非在 TEX 输出完毕后最后输出图像要重新缩减回去, 或者你在实验一种特殊效果。

放大字体的另一种方法是相对于设计尺寸给出放大比例因子。例如, 命令

```
\font\magnifiedfiverm=cmr5 scaled 2000
```

是得到 `cmr5` 字体两倍尺寸的另一种方法。放大比例因子规定是整数, 等于放大比例乘以 1000。因此, 放大因子为 1200 就是放大 1.2 倍, 等等。

▶ **练习4.6**

用两种方法把 `cmr10` 的正常尺寸的一半大小的字体载入 TEX 内存中。

在许多计算机中心, 已表明按照几何比例使用放大的字体是很有益的——有点象钢琴上精确的调音。这个思想使所有字体在正常尺寸, 1.2 倍和 1.44(1.2 × 1.2) 倍尺寸都可以使用; 可能还可以放大到 1.728(= 1.2 × 1.2 × 1.2) 倍或更大。因此, 你可以把整个书稿放大 1.2 或 1.44 倍而仍然得到精美的字体。Plain TEX 提供了一个缩略命令, `\magstep0` 是放大 1000 比例因子, `\magstep1` 是放大 1200 比例因子, `\magstep2` 是 1440, 如此直到 `\magstep5`。例如, 使用

```
\font\bigtenrm=cmr10 scaled\magstep2
```

就载入了 `cmr10` 字体的正常尺寸的 1.2 × 1.2 倍。

"This is `cmr10` at normal size (`\magstep0`)."
"This is `cmr10` scaled once by 1.2 (`\magstep1`)."
"This is `cmr10` scaled twice by 1.2 (`\magstep2`)."

(注意: 略微放大这种方法用处很大。) 还有一个命令 `\magstephalf`, 它放大 $\sqrt{1.2}$ 倍, 即介于 `\magstep0` 和 `\magstep1` 之间。

除了字体载入时已经被给定的放大之外, 第10章阐述了怎样在整个书稿上使用放大方法。例如, 如果你已按 `\magstep1` 载入一个字体, 并且你还规定 `\magnification=\magstep2`, 那么在打印时所用的实际字体就按 `\magstep3` 放大。类似地, 如果载入字体的放大是 `\magstephalf` 且 `\magnification=\magstephalf`, 那么打印结果放大 `\magstep1`。

5. 编组

不时地需要把文稿的一部分当成一个整体来处理, 所以你应该声明此部分的开头和结尾。为此, TEX 给两个"编组字符"赋予了特殊含义, 它(象转义符一样)与所键入的正常符号的处理方法不同。在本手册中我们设定 { 和 } 是编组字符, 因为在 plain TEX 中如此。

在前一章, 我们已经遇见了编组的例子, 那里用到的是在组中的字体变化不影响外面的字体。就象外面后面要学习的一样, 同一原理可以应用到定义在组中的几乎所有内容; 例如, 如果你在某些组中定义了一个控制系列, 当组结束时, 那些定义就消失了。用这种方法, 通过在组中临时改变其正常规则, 可简便地告诉 TEX 完成特殊的工作; 因为这种变化在组外部将消失了, 所以当特殊工作结束后, 不必担心文稿后面的部分怎样才能恢复正常规则。因为编组一般在编程语言方面是很重要的, 所以计算机方面上它有一个名字, 即"模块结构", 只在组中起作用的定义相对于那个组是"局部的"。

即使你不关心模块结构, 就是为了更好地得到间距, 你可能也要利用编组。例如, 让外面再讨论本手册中得到标识符'TEX'的控制系列 `\TeX`: 在第三章我们看到, 如果不键入'`\TeX\ `', 控制系列后面的空格会被吃掉, 但是当后面跟的是非空格字符时, 键入'`\TeX\`'又出现错误。噢, 在所有情况下, 给出一个简单的组:

```
{\TeX}
```

总是正确的, 不管跟着的字符是不是空格, 因为 } 使 TEX 不能把可能的空格吃掉。当你正在使用文本编辑器时这更方便(比如, 当用控制下列代替所有出现的特殊单词时)。你也可用另一种方法, 就是键入

```
\TeX{}
```

用空组来达到同样的目的: 这里的'`{}`'是没有字符的组, 所有它没有输出结果, 但是它的确有阻止 TEX 吃掉空格的效果。

▶ **练习5.1**
有时候, 会遇到一个类似'shelfful'的罕见单词, 不用'ff'连字符而用'shelfful'看起来更好。想想怎样愚弄一下 TEX, 让它认为这样的单词中的两个 f 不是紧接着的。


▶ **练习5.2**
不用'\␣', 看看怎样在一排中得到 3 个空格?

但是 TEX 也在另外非常不同的方面使用编组, 即规定特定的控制系列作用在哪些文本上。例如, 你要把一行中的内容居中, 利用定义在 plain TEX 中的控制系列 `\centerline`, 你可以键入

```
\centerline{This information should be centered.}
```

编组还出现在 TEX 的很少几个更复杂的指令中; 当浏览附录 B 时你就会看到, 可以组中套组。但是在普通文稿中, 一般不需要复杂的编组, 所以你不必担心。只是别忘了把每个开始的组要结束掉, 因为忘记'}'可能会出问题。

这里有一个两个组的例子, 一个嵌套在另一个中间:

```
\centerline{This information should be {\it centered}.}
```

正如所预期的一样, TEX 产生了一个居中的行, 其中包含 italic 字体:

This information should be *centered*.

但是, 让我们更详细地讨论这个例子: '`\centerline`'出现在大括号外面, 而'`\it`'出现在里面。为什么这两种情况不同呢? 初学者怎样才能弄清谁是谁呢? 答案是: `centerline` 这个控制系列是只作用在其后接下来的内容上, 所以你要把居中的内容放在括号中(除非其内容仅仅是一个符号或一个控制系列)。例如, 要把标识符 TEX 居中, 键入 '`\centerline\TeX`' 就可以了, 但是要居中短语'TEX has groups', 就需要括号:'`\centerline{\TeX\ has groups}`'。另一方面, `\it` 这个控制系列简单表示"改变当前字体"; 它对前面没有作用, 所以它——至少暗中——对其后的所有内容起作用。括住 `\it` 的括号是为了把字体改变的作用限制在局部区域。

换句话说, 本例中的两组括号的作用不同: 一个是把文本的多个单词看作一个单一对象, 而另一个提供了局部模块结构。

▶ **练习5.3**

如果你键入下列内容, 会得到什么结果?

```
\centerline{This information should be {centered}.}
\centerline So should this.
```

▶ **练习5.4**

下面这个呢?

```
\centerline{This information should be \it centered.}
```

▶ **练习5.5**

定义一个控制系列 `\ital`, 使得用户用键入'`\ital{text}`'来代替'`{\it text\/}`'。与 `\it` 相比, 看看 `\ital` 有什么优缺点。

后面的章节规定了许多 TEX 的原始操作, 其中编组的局部性很重要。例如, 当 TEX 的一个内部参数在组中变化了, 当组结束时, 此参数会恢复为原先的内容。但是有时候, 希望一个定义能不限在当前组中。这可以通过在定义上加前缀'`\global`'而实现。例如, TEX 把当前的页码放在叫做 `\count0` 的寄

存器中, 并且输出一页的程序要增大页数。总是通过把输出程序放在组中来保护它们, 使得它们不会混淆 TEX 的其它内容; 但是如果对输出组而言 \count0 是局部的, 那么产生的变化就会被掩盖。命令

```
\global\advance\count0 by 1
```

就解决了这个问题; 它增大 count0, 并且使其值在输出程序结束时保留下来。 一般地,\global 使其后紧接的定义适用于所有存在的组, 而不仅仅是最内层的组。

▶ **练习5.6**

如果你认为自己掌握了局部和整体的定义, 这里有一个小测验: 设 \c 代表'\count1=', \g 代表'\global\count1=', \s 代表'\showthe\count1'。下面的结果是什么?

```
{\c1\s\g2{\s\c3\s\g4\s\c5\s}\s\c6\s}\s
```

利用 TEX 得到模块结构的另一种方法是用原始的 \begingroup 和 \endgroup。这些控制系列很容易在一个控制系列中开始一个组, 并且在另一个控制系列中结束组。在控制系列展开后, TEX 实际执行的文本必须是真正的嵌套组, 即, 组不能交叉。例如,

```
{ \begingroup } \endgroup
```

是不合法的。

▶ **练习5.7**

定义一个控制系列 \beginthe⟨block name⟩ 和 \endthe⟨block name⟩, 它给出一个"命名"模块结构。 换句话说,

```
\beginthe{beguine}\beginthe{waltz}\endthe{waltz}\endthe{beguine}
```

是允许的, 但是不允许

```
\beginthe{beguine}\beginthe{waltz}\endthe{beguine}\endthe{waltz}.
```

6. 运行 TeX

掌握 TeX 的最好方法就是使用它。因此, 对你而言, 该坐在终端前与 TeX 系统交流交流, 试一下看看到底怎么样。首先为你准备的是一些短小而全面的例子。

注意: 本章相当长。要不要现在休息一下, 明天接着看?

现在, 假定你已经精力充沛, 并且急于进行 TeX 实战。逐步使用它的指导都在本章中。首先要做的是: 去有图像输出设备的实验室, 因为你将要看到所得到的结果——从远程运行 TeX 不那么过瘾, 因为你不能亲手拿到生成的文档。于是, 登录; 运行 TeX。(可能你必须问问别人看看怎样做。通常操作系统提示符出现后, 键入'`TeX`'或'`run tex`'或类似的命令。)

当你成功运行后, TeX 将出现如下欢迎信息:

```
This is TeX, Version 3.14 (preloaded format=plain 89.7.15)
**
```

'`**`'是 TeX 要求你输入文件名。

现在, 键入'`\relax`'(要包含反斜线)和 ⟨return⟩ (或者其它在你的终端上表示一行结束的符号)。TeX 已经做好准备以读入一篇长文稿; 但是你刚才的意思是简单做点事, 因为仅仅运行了一下。实际上, `relax` 是一个控制系列, 意思是"do nothing"。

计算机将输出一个星号来回应你。现在键入象'`Hello?`'这样的内容并且等待下一个星号。最好键入'`\end`'且等一下看看有什么情况。

TeX 将回应出'`[1]`'(意思是它完成了一页输出); 接着程序将停止, 可能还有一些显示说它生成了一个叫'`texput.dvi`'的文件。(如果你在输入的第一行没有给定名字, TeX 将使用 `texput` 这个名字; 还有, `dvi` 的意思是"不依赖于设备", 因为 `texput.dvi` 可以在几乎所有的印刷输出设备上打印出来。)

现在你又需要周围计算机高手的帮助了。他们会告诉你怎样把 `texput.dvi` 打印出来。当你看到结果——噢, 太漂亮了!——你会看到一个漂亮的'Hello?', 并且在底部中间有页码'1'。为你的精美印刷的处女作干杯。

关键是你现在知道了实现它的整个过程。剩下的就是用同样的方法处理更长的文稿。所以我们接下来用文件举例, 而不是逐行输入。

用你喜欢的文本编辑器建立一个文件, 名字为 `story.tex`, 它包含下列 18 行文本(不多不少):

```
\hrule
\vskip 1in
\centerline{\bf A SHORT STORY}
\vskip 6pt
\centerline{\sl by A. U. Thor}
```

```
6 \vskip .5cm
7 Once upon a time, in a distant
8   galaxy called \"O\"o\c c,
9 there lived a computer
10 named R.~J. Drofnats.
11
12 Mr.~Drofnats---or ``R. J.,'' as
13 he preferred to be called---
14 was happiest when he was at work
15 typesetting beautiful documents.
16 \vskip 1in
17 \hrule
18 \vfill\eject
```

(当然, 别键入这些行左边的数字; 它们只起参照的作用。) 这个例子有点长, 而且还有点无聊; 对象你这样优秀的排版者没有窍门, 并且它将为你提供宝贵的经验, 所以要完成它。为了自己! 键入时想一想; 随着你键入这个文件, 这个例子出现了你可能已经掌握的几个 TEX 的重要特征。

这里对你刚刚键入的内容做一个简要的解释: 第 1 和 17 行放了一根贯穿整页的水平线(细线)。第 2 和 16 行跳过了一英寸的空白; '`\vskip`'表示"垂直跳跃", 并且这个额外的空白将把水平线与文档的其它内容分开。第 3 和 5 行生成了题目和作者的名字, 居中且用 bold 字体和用 slanted 字体。第 4 和 6 行在相应行和随后的行之间放上空白。(我们将在第十章讨论象'`6pt`'和'`.5cm`'这样的测量单位。

本 story 的主体在第 7–15 行, 并且它包含两个段落。第 11 行是一个空行就告诉 TEX, 第 10 行是第一个段落的最后一行; 还有, 第 16 行的'`\vskip`'意味着第二段到第 15 行结束, 因为垂直跳跃不出现在段落中。顺便说一下, 本例子看起来含有丰富的 TEX 命令; 但是在那方面它不是典型的, 因为它太短以及被用来做教学。象 `\vskip` 和`\centerline`这样的结构可能应该放在文稿的开头, 除非你在用一个刻板的格式, 但是它们不会太多; 在大部分的时间你都在键入规整的文本以及相对少量的控制系列。

现在, 如果你以前不曾用过计算机排版, 那么你有福了: 在一个段落中, 你不必关心在什么地方断行(即在右边的哪里结束本行并重新开始一行), 因为 TEX 将替你做。你的文稿文件可以包含长行和/或短行; 这不是问题。当你修改时这特别有用, 因为除了单词外你不必改动任何东西。*在你的文稿中, 每个新行本质上相相当于一个空格。* 当 TEX 读完整段时——在本例中就是第 7 到 11 行——它将设法把文本断开, 使得除最后一行外的每行输出包含同样多的材料; 并且为了保持间距的一致性, 在有必要的地方加入连字符, 但这只是最后一招。

第 8 行含有怪怪的东西

```
\"O\"o\c c
```

并且你已经知道, \" 表示一个变音的重音。\c表示一个“变音符号”, 所以你得到了‘Ööç’来作为那个遥远银河的名字。

剩下的文本就是以前我们讨论过的破折号和引号的用法, 只有在第 10 和 12 行的‘~’符号是个的波纹号。它们称为带子, 因为它们的单词绑在一起; 即 TEX 把‘~’看作正常的空格, 但是不能在那里断行。好的排版应该在名字之间用带子, 就象我们的例子一样; 关于带子更多的讨论见第十四章。

最后, 第 18 行告诉 TEX 要‘\vfill’, 即, 用空白把本页其余的部分填满; 接着‘\eject’本页面, 即把它送到输出文件。

现在进行第二次实战: 再次运行 TEX。这次当计算机出现‘**’时, 你应该键入‘stroy’, 因为那是你的输入资料所在的文件的名字。(文件也可以用它的全名‘story.tex’, 但是如果后缀没有给出, TEX 将自动加上后缀‘.tex’。)

你可能想知道为什么第一个提示符是‘**’, 而后面的是‘*’; 原因很简单, 你所键入到 TEX 的第一个东西与其余的略有不同: 如果在‘**’后你键入的第一个字符不是反斜线, 那么 TEX 将自动插入‘\input’。因此, 通常只要命名了输入文件, 你就可以运行 TEX 了。(目前的 TEX 系统要求键入‘\input story’而不是‘story’来开始处理, 并且你仍然可以那样做; 但是大多数 TEX 用户喜欢把它们所有的命令都放在一个文件中, 而不是逐行键入, 所有现在 TEX 不需要每次开始都键入 \input 来惹人厌烦。) 回想一下, 在第一次实战中, 你键入了‘\relax’, 它以反斜线开头, 所以 \input 没有暗中插入。

实际上在‘**’和‘*’之间有另外一个差别: 如果 ** 后的第一个字符是 ampersand (‘&’), TEX 将在处理之前用预编译的格式文件来替换其内存。于是, 比如, 你可以在‘**’后键入‘&plain \input story’或仅仅是‘&plain story’, 如果你正在运行没有预载入的 plain 格式的某些 TEX 版本。

顺便说一下, 很多系统都允许你通过在命令行键入象‘tex story’这样的命令来调用 TEX, 而不用等‘**’; 类似地, ‘tex \relax’就是实战一, ‘tex &plain story’在输入 story 文件前载入了 plain 格式。你可能想试试, 看看它在你的计算机上是否可行, 或者你可能会问别人是否有一个类似的快捷方法。

随着 TEX 开始读入你的 story 文件, 它在终端输出了‘(story.tex’, 可能会有精确识别的版本号, 这依赖于你的操作系统。接着, 它输出了‘[1]’, 意思是第 1 页处理完毕; 接着是‘)’, 意味着文件被全部读入了。

TEX 现在又用‘*’提示你了, 因为文件中不包含‘\end’。现在输入 \end, 就得到一个文件 story.dvi, 它包含 Thor 的 story 排版后的结果。就象在实战一中一样, 你可以把 story.dvi 打印出来; 现在去做吧。这里没有给出排版输出, 但是你可以通过亲自实战得到结果。请在继续之前完成它。

▸ **练习6.1**

统计表明, 在读本手册的人中, 10 个中有 7.43 个实际上按照建议键入了 `story.tex` 文件, 但是那些人掌握 TEX 却最好。难道你不想变成他们那样?

▸ **练习6.2**

仔细观察实战二的输出, 把它与 `story.tex` 相比较: 如果你认真阅读了用法说明, 你会发现一个排版错误。它是什么? 为什么它会偷偷出现在那里?

现在, 你知道怎样从一个文件得到文档了。本章剩下的实战要帮助你应付后面运行中不可避免的异常情况; 我们将有意用一些方法使 TEX “难受得只叫”。

但是在继续之前先把前面输出的错误纠正过来(见 exercise 6.2): 文件 story.tex 的第 13 行应当改为

```
he preferred to be called---% error has been fixed!
```

这里的‘%’号是 plain TEX 中我们以前没讨论过的特征: 它使得输入文件的行有效地终止而不引入换行时 TEX 通常要插入的空格。还有, TEX 将忽略掉 % 后的任何内容, 直到文件的那行的结尾, 这样就可以在文稿中加入注释, 这些注释只是为了阅读方便。

实战三将让 TEX 处理更吃力, 这是通过要求它把 story 放在越来越窄的栏中。如下操作: 启动程序后, 在‘**’后键入

```
\hsize=4in \input story
```

这意味着“把 story 放在 4 英寸的栏中”。更准确地说, `\hsize` 是 TEX 的一个原始控制系列, 它给出了水平宽度, 即当排版段落时输出中每个正常行的宽度; 还有, `\input` 也是一个原始控制系列, 它使 TEX 读入给定的文件。于是, 你告诉计算机改变由 plain TEX 定义的 `\hsize` 的正常设置, 并且在这种修改后的设置下处理 `story.tex`。

如前 TEX 输出类似‘`(story.tex [1])`’后, 接着出现‘`*`’。现在你应该键入

```
\hsize=3in \input story
```

接着, TEX 输出‘`(story.tex [2])`’后又出现‘`*`’, 键入下面三行

```
\hsize=2.5in \input story
\hsize=2in \input story
\end
```

这就完成了这个四页的实战。

在处理 2-inch 宽度时, TEX 会发出几次‘`Overfull \hbox`’警告, 别慌; 这就是在实战三中设定的问题。在把给定的段落分解为正好 2 英寸宽度时, 如果不允许单词间的间距太大或太小, 是没

有简单的好办法的。Plain TEX 已经对所有行设置了一个相当严格的公差, 由它得到

you don't get spaces between words narrower than this, and
you don't get spaces between words wider than this.

如果没有办法满足这些限制, 就得到了一个溢出的盒子。对于这个溢出的盒子, 还出现 (1) 警告信息, 出现在终端上, 和 (2) 在溢出盒子的右边插入一个大黑块, 出现在输出结果中。(看看实战三输出的第四页; 溢出的盒子象大拇指一样伸出来。 另一方面, 第一到三页就很漂亮。)

当然, 你不想在输出中有溢出的盒子, 所有 TEX 给出几种方法来去掉它们; 这就是我们实战四的主题。但是首先要仔细观察实战三的结果, 因为当 TEX 被迫使用溢出的盒子时, 给出了一些有价值的信息; 你应该知道怎样分析这些数据:

```
Overfull \hbox (0.98807pt too wide) in paragraph at lines 7--11
\tenrm tant galaxy called []O^^?o^^Xc, there lived|
Overfull \hbox (0.4325pt too wide) in paragraph at lines 7--11
\tenrm a com-puter named R. J. Drof-nats. |
Overfull \hbox (5.32132pt too wide) in paragraph at lines 12--16
\tenrm he pre-ferred to be called---was hap-|
```

每个溢出的盒子都与输入文件中的相应位置有关(比如, 当处理 `story.tex` 的第 7–11 行时出现前两个警告), 并且还可以知道伸出了多少宽度(比如, 0.98807 points)。

注意, TEX 还用简略的形式给出了溢出的盒子的内容。例如, 最后一个警告有这样的信息:'he preferred to be called—was hap-', 设定字体为 `\tenrm` (10-point roman 字体); 第一个所给出的内容是有些古怪的'Öõç'的表述, 因为重音出现在字体的奇怪的地方。一般地, 当你在这些信息中发现'`[]`'时, 就表示段落缩进或某些复杂的指令; 在这种特殊情况下, 它表示一个'O'上的重音符号。

▶ **练习6.3**
请解释一下出现在那个信息中'`lived`'后面的那个'`|`'。

▶ **练习6.4**
为什么在'`Drof-nats. |`'中的'`|`'前有一个空格?

你不必在溢出盒子的信息在屏幕上消失前在纸上记下这些信息, 因为 TEX 总是把它们写成记录或"log 文件", 它们记下了每个活动期间所发生的事情。例如, 现在你有一个文件叫 `story.log`, 它包含实战三的记录, 以及文件 `texput.log`, 它包含实战一的记录。(当你进行实战三时, 实战二的记录可能已经被覆盖掉了。) 现在看一下 `story.log`; 就会看到伴随溢出盒子的信息的不但有简略的盒子的内容, 而且有某些与 hbox 和 glue 以及 kern 等等有关的奇怪的信息。

这些资料给出了溢出盒子的精确描述; 在TEX 奇才们被请求诊断某些不可理解的错误时, 将发现这些罗列信息的重要性, 并且你可能也有一天想要懂得 TEX 的内部代码。

溢出盒子的简略形式给出了 TEX 在溢出之前在连字化方面的尝试。连字算法阐述在附录 H, 它是优秀的但并不完美; 例如, 从 `story.log` 的信息中你可以看到, TEX 找到了'pre-ferred'中的连字, 并且它可以把'Drof-nats'连字化。但是它发现'galaxy'中没有连字符, 有时候, 简单地通过给 TEX 一个线索, 让它知道怎样更完整地把某些单词连字化, 可以解决盒子的溢出问题。(后面我们将看到, 有两种方法来实现这个功能: 象在'`gal\-axy`'中每次插入任意连字符, 或者通过在文稿开始声明一次'`\hyphenation{gal-axy}`'。)

在当前的例子中, 连字符不是一个问题, 因为 TEX 找到并尝试了所有可能有用的连字符。解决盒子溢出的唯一方法是改变公差, 即在单词之间允许更大的间距。的确, plain TEX 对宽行设定的公差与 2-inch 的栏完全不相称; 不放松这些限制, 这么窄的栏就不能完成任务, 除非你为适应它而重新改写文稿。

为了评估间距的品质, TEX 为它设置的每行规定了一个数值叫做"badness"。"badness"的严格规则因字体不同而不同, 它们将在第十四章中讨论; 但是, 这里给出了 plain TEX 中 roman 字体的 badness 的情况:

The badness of this line is 100.	(very tight)
The badness of this line is 12.	(somewhat tight)
The badness of this line is 0.	(perfect)
The badness of this line is 12.	(somewhat loose)
The badness of this line is 200.	(loose)
The badness of this line is 1000.	(bad)
The badness of this line is 5000.	(awful)

Plain TEX 一般规定, 行的 badness 不可大于 200; 但是在现在的情形下, 因为

'tant galaxy called Ööç, there'	has badness 1521;
'he preferred to be called—was'	has badness 568.

所以无法满足要求。因此我们现在进行实战四, 其中对窄栏采用更适当的间距的变化。

再次运行 TEX, 这次在开头键入

```
\hsize=2in \tolerance=1600 \input story
```

使得要容许的 badness 放宽到 1600。万岁! 这次没有盒子溢出了。(但是你却得到一个松散的盒子, 因为如果盒子的 badness 超过一个叫 `\hbadness` 的阈值, TEX 也给出警告; plain TEX 设置

\hbadness=1000。) 现在键入

```
\hsize=1.5in \input story
```

让 TEX 更难处理, (这样就是保持容许度为 1600 而栏的宽度更窄)。唉, 盒子又溢出了; 为了看看出现什么问题, 那么再试试

```
\tolerance=10000 \input story
```

TEX 把 10000 看作"无穷大"的容许度, 允许任意宽的间距; 因此, 容许度为 10000 将永远不出现盒子溢出, 除非比栏自己的宽度还宽且无连字化的单词出现。

在 1.5-inch 情形下, TEX 出现的松散盒子实在难看; 在这么窄的极限下, 偶尔出现宽间距是不可避免的。而思路一变, 试试

```
\raggedright \input story
```

(它告诉 TEX 不用对齐右页边, 且在每行中保持间距一致。) 最后, 键入

```
\hsize=.75in \input story
```

再接着键入'\end'以结束实战四。这使得栏变得出奇地窄。

在把段落分成大致相等的行这个问题上, 本实战的输出结果会让你有所体会。当行相对宽时, TEX 几乎总能找到好的解决方法。否则你必须给出一些折衷的方案, 可能有几种可选择的方法。假设你要保证行的 badness 不超过 500。那么你可以把 \tolerance 设为较大的值, 以及 \hbadness=100; TEX 不出现溢出的盒子, 但是会对松散的盒子发出警告。或者你设置 \tolerance=500; 那么 TEX 可能出现溢出的盒子。如果你真希望起到提示纠正的作用, 第二个办法更好, 因为你可以看到溢出的盒子伸出了多少; 对可能的补救就一目了然了。另一方面, 如果你没有时间来处理这些难看的间距——如果你只想看看它难看在哪里——那么第一种方法更好, 只是会花费计算机更多的时间。

▶ **练习6.5**
当设定为 \raggedright 后, badness 反映的是右页边的空白的量, 而不是单词间的间距。设计一个方法, 在 story 设置为 1.5-inch 栏且 ragged-right 时, 通过它可以很容易得到 TEX 给每行计算出的 badness。

有一个叫 \hfuzz 的参数, 它允许不理会那些只略微溢出的盒子。例如, 如果设置 \hfuzz=1pt, 那么只有超出部分大于 1 point 的盒子才被认为是不正确的。Plain TEX 的设置是 \hfuzz=0.1pt。

▶ **练习6.6**
仔细观察实战四的输出, 特别是第三页, 可以发现, 在窄栏情况下, 只要同一行的其它间距变大, 在破折号前后允许出现空白效果会更好。定义一个叫 \dash 的宏来实现本要求。

已经提醒过你: 本章很长。但是要记住: 还只有一个实战, 以后在运行 TEX 时你就胸有成竹了。你还不知道怎样处理错误信息——即, 不仅仅是象盒子溢出这样的警告, 而是 TEX 停下来要求你干预的情况。

当你没有思想准备时, 错误信息可能有些可怕; 但是当你端正了态度后, 它们可能还很有意思。只是要记住, 你其实别怨计算机, 而且没人用这些错误来攻击你。接下来你会发现, 运行 TEX 实际上可能是一次创造体验, 而不是可怕的老虎。

实战五的第一步是在文件 story.tex 中故意放两个错误。把第 3 行改为

```
\centerline{\bf A SHORT \ERROR STORY}
```

并且把第 2 行的'\vskip'改为'\vship'。

再次运行 TEX; 但是不键入'story', 而键入'sorry'。计算机就回答说它找不到文件 sorry.tex, 并且它会要求你再尝试一下。这次, 只单击 ⟨return⟩; 将看到你最好给出一个真实文件的名字。因此, 键入'story'并等 TEX 找到那个文件中的过失中的一个。

噢, 计算机很快停下来, (注: 某些 TEX 的安装不支持交互模式。在这种情况下, 你只能在 log 文件中查看错误信息, 在那里, 这些错误信息和"help"信息一起出现。) 输出如下信息:

```
! Undefined control sequence.
l.2 \vship
           1in
?
```

TEX 以'!'开始错误信息, 并且它通过给出两行上下文来提示错误出现时的内容。上面一行(在本例是'\vship')给出了 TEX 目前读的内容, 并且告诉了它的位置('l.2',即第 2 行); 下面一行(本例中的'1in')是 TEX 将要读入的内容。

在显示的上下文后面出现的'?'意味着 TEX 要求给出下一步怎样做。如果你以前没有见过错误信息, 或者你忘记应该用什么来应答, 现在你可以键入'?'(继续来试试吧!); TEX 将给出下列信息:

```
Type <return> to proceed, S to scroll future error messages,
R to run without stopping, Q to run quietly,
I to insert something, E to edit your file,
1 or ... or 9 to ignore the next 1 to 9 tokens of input,
H for help, X to quit.
```

这是供你选择的清单。你可以用各种方法来选择进行运行:

1. 直接键入 ⟨return⟩. 在尽可能修复错误后, TEX 将恢复运行。
2. 键入'S'。如果还有错误出现, TEX 将直接运行, 不再提示。随后的错误信息将在你的屏幕上闪过, 快到你来不及看它们, 而且它们将出现在你的 log 文件中, 在那里你可以静下心来细细查看。所以, 'S'就象对每个错误都键入 ⟨return⟩。

3. 键入'`R`'。这类似于'`S`', 但是更强大, 因为它告诉 TEX, 不管什么原因都不停止, 即使文件名都找不见。
4. 键入'`Q`'。它类似于'`R`', 但还要强大, 因为它不但告诉 TEX 无停顿地运行, 而且不再在屏幕上输出信息。它是一种快速但不考虑后果的运行方式(就是放任 TEX 去运行而不进行任何干预)。
5. 键入'`I`', 后面跟一些你要插入的文本。TEX 将首先读入此行再继续读入原来要读入的内容。用这种方法插入的行不以空格为结尾。
6. 键入一个数(小于 100)。TEX 将从接下来读入的字符和控制系列中删除这么多字节, 并且做完后它会再停下来等待干预。
7. 键入'`H`'。这就是你现在所做的事情, 对你暂时没搞清楚的任何错误信息都可以使用。对发现的每个错误, TEX 都给出两个信息: 正式的和非正式的。 正式的信息首先输出(比如, '`! Undefined control sequence.`'); 如果你键入'`H`'请求更多帮助, 非正式的信息才输出, 而且, 如果错误信息翻滚过去了, 你还可以在 log 文件中查看。非正式的信息尝试着给出 TEX 所认为的问题所在来补充正式的信息, 并且通常为修补错误而提出一个方法。
8. 键入'`X`'。它表示要"exit"。在把已完成的相关任务写入 `log` 文件和已输出的页面后, 它把 TEX 终止。当前(未完成的)页面不产生输出。
9. 键入'`E`'。它类似于'`X`', 但是它还告诉计算机对当前文件进行编辑, 位置就是当前停止的位置, 这样你就可以很方便修改后再次运行。

当你键入'`H`'(或者'`h`')后, 就得到一个提示, 它尝试着解释说, TEX 刚刚读入的控制系列(即, `\vship`)没有给出定义, 并且你要么插入一个正确的控制系列, 要么直接继续, 就象这个讨厌的东西没有出现过一样。

在本情形下, 你最好的选择是键入

```
I\vskip
```

(并且 ⟨return⟩), 注意, 在'`I`'后面没有空格; 这就把 `\vship` 替换成了 `\vskip`。(试试吧!)

如果你直接键入 ⟨return⟩ 而不是其它选项, 那么 TEX 将继续运行并读入'`1in`', 它将被看作要排版的段落的一部分。你也可以键入'`3`'; 它将从 TEX 的输入中删除掉'`1in`'。另外你也可以键入'`X`'或'`E`'以改正文件中的拼写错误。但是一般最好是每次运行 TEX 时检出尽可能多的错误, 这样可以提高效率和减少成本。第二十七章阐述了引导 TEX 处理这些出问题的文本的更多技术。



▶ **练习6.7**

如果你在 `\vship` 这个错误后键入'`5`', 会出现什么情况?



通过在文件以及行上给定命令, 你可以控制交互的等级: TEX 的原始 `\scrollmode`, `\nonstopmode`, 以及 `\batchmode` 分别对应于对错误信息键入'`S`', '`R`', 或'`Q`', 而且 `\errorstopmode` 转换回正常的交互等

级。(这样的改变是全局的, 不管它们是否出现在组中。) 还有, 很多已安装好的系统已经设置了 TEX 运行时中断它的方法; 这样的中断使程序转换回 `\errorstopmode`, 在此处 TEX 停下来等待干预。

实战五接下来会出现什么呢? TEX 将被噎在我们故意设置的另一个缺陷上。但是这次, 错误信息更详细, 因为出现了 6 行而不是两行:

```
! Undefined control sequence.
<argument> \bf A SHORT \ERROR
                              STORY
\centerline #1->\line {\hss #1
                              \hss }
l.3 \centerline{\bf A SHORT \ERROR STORY}

?
```

当 TEX 在处理某些高级命令而检测到的错误时, 你就会得到类似的多行错误信息——在本例中, 它正在处理非原始的定义 `\centerline`(它定义在 plain TEX 中)。首先, 这样的错误信息对你毫无用处, 因为你所看到的是不曾写过的低级 TEX 代码。但是通过体会 TEX 的运作方式, 你会处理好这个错误。

首先注意到, 上下文的内容总出现在两行中。象以前一样, 上一行表示 TEX 刚刚读入的(‘`\bf A SHORT \ERROR`’), 接着出现的是将要读入的(‘`STORY`’)。 接下来的两行是上面两行的上下文; 它表明, 在开始读入上面两行前 TEX 所做的事情。在本例我们看到, TEX 刚刚读入‘`#1`’, 它是一个特殊代码, 来告诉计算机“读入由当前控制系列处理的第一个变量”; 即“现在读入 `\centerline` 要居中的行的内容”。在附录 B 中的定义声明, 当应用到某些文本时, `\centerline` 要做的是用那些文本来代替‘`\line{\hss#1\hss}`’中的‘`#1`’。所以 TEX 运行到了 `\centerline` 的展开代码的中间代码, 即要居中的文本的内容。

底部的行表示到现在为止 TEX 得到的文件 `story` 中的进度。(实际上, 本例的底行是空的; 出现在底行的实际上是第一个两行的上下文, 并且它表明 TEX 已经读入包括‘`}`’的文件的第 3 行的所有内容。) 因此, 在本错误信息中的内容让我们一览 TEX 的处理方式。首先, 它遇见第 3 行开头的 `\centerline`。接着它得到 `\centerline` 的定义, 并注意到 `\centerline` 有一个“变量”, 即, `\centerline` 要应用在跟着的下一个字符或者控制系列, 或者组上。因此 TEX 继续读入, 把‘`\bf A SHORT \ERROR STORY`’提出来作为 `\centerline` 的变量。接着它开始展开定义, 正如附录 B 所定义的一样。当它读到 `#1`, 就开始读入存储的变量。并且当遇到 `\ERROR` 时就给出未定义的控制系列这个错误信息。

▶ **练习6.8**

为什么 TEX 第一次遇到 `\ERROR` 时(即在读第 3 行的‘`STORY`’之前)不认为它是一个未定义的控

制系列呢?

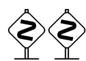
当你看到类似的错误信息时, 找到问题根源的最好思路通常是底行(因为它是你键入的内容)和顶行(因为那里是错误发生的地方)。在那里的某个地方你通常可以发现问题。

现在你该怎么办? 如果现在键入'H', 得到的是与你以前看到的一样的关于未定义控制系列的帮助信息。如果键入 ⟨return⟩, TEX 将继续运行, 得到与实战二相同的输出结果。换句话说, 一般的干预没什么新的东西了。那么, 键入'E'; 它将终止运行并打开有错误的文件。(在某些系统上, TEX 实际上启动一个标准的文本编辑器, 并且定位在要删除'\ERROR'的地方。在其它系统上, TEX 将直接告诉你编辑文件 story.tex 的第 3 行。

当你再次编辑 story.tex 时, 会注意到第 2 行包含的还是 \vship; 事实上, 你要 TEX 插入 \vskip 并不意味着文件有任何改变。一般地, 在运行时, 你应该改正 TEX 发现的所有错误; log 文件为记录那些出现错误提供了一种方便的方法。

好了, 本章的确比较长, 那么我们总结一下。通过上面的五个实战, 你已经直接掌握了: (1). 怎样通过 TEX 输出一个的文稿; (2). 怎样制作包含这个 TEX 文稿的文件; (3). 怎样改变 plain TEX 的格式以得到不同宽度的栏; (4). 当 TEX 给出警告时怎样有条不紊地修正。

那么, 你现在可以不再阅读本书, 并且可输出很多文档了。但是最好继续下去, 因为你正处在深造的关头。而且至少你应该读读第七章, 因为它给出了不必键入的一些符号, 除非你想要 TEX 做某些特殊的事情。当阅读剩下的章节时, 当然最好做一些实战练习, 你自己去设计实验吧。



如果你使用别人设计的 TEX 格式包, 错误信息可能包括许多不可预测的双行的宏的内容。通过在你的文件开头设置 \errorcontextlines=0, 可以减少输出的信息的量; TEX 将只输出最上面和最下面两行内容, 以及与 \errorcontextlines 有关的附加的两行内容。(如果因此忽略了一些内容, 你还会看到'...'。) 如果当大量错误信息被去掉时仍能找到错误所在, 就太幸运了; 否则, 你可以键入'I\errorcontextlines=100\oops'并且再次运行。(它一般给出了未定义控制系列的错误和大量上下文。)

7. TEX 工作原理

在上一章我们看到, 输入的文稿用"行"的方式来表达, 但是这些输入的行和最后出现在页面中的行本质上是不同的。因此, 当输入或编辑文件时, 你可以随心所欲地换行。这里要提出几个相关的规则:

- 一个 ⟨return⟩与一个空格类似。
- 一行中的两个空格看作一个。
- 一个空行表示一段的结束。

严格说, 这些规则是矛盾的: 在一行内键入两次 ⟨return⟩ 就得到一个空行, 而且这与在一行键入两个空格不同。有一天你可能想了解真正的规则。在本章和下一章, 我们将讨论从输入到输出的相当初始的阶段。

首先, 明智之举是对键盘发送到计算机中的准确内容有一个了解。TEX 可能见到的直接从键盘上键入文件或行中的字符有 256 个。这 256 个字符被分为 16 类, 数字是从 0 到 15:

(类)*Category*	意义	
0	转义符	(在本手册为 \)
1	组开始	(在本手册为 {)
2	组结束	(在本手册为 })
3	数学环境	(在本手册为 $)
4	表格对齐	(在本手册为 &)
5	换行	(在本手册为 ⟨return⟩)
6	参数	(在本手册为 #)
7	上标	(在本手册为 ^)
8	下标	(在本手册为 _)
9	可忽略的字符	(在本手册为 ⟨null⟩)
10	空格	(在本手册为 ␣)
11	字母	(A, . . . , Z 和 a, . . . , z)
12	其它字符	(不在上下文的其它字符)
13	活动符	(在本手册为 ~)
14	注释符	(在本手册为 %)
15	无用符	(在本手册为 ⟨delete⟩)

对你而言, 不必掌握这些代码数; 要点只是 TEX 对 16 类不同字符所做的处理。首先, 本手册中出现的只有两种类型——转义符和其它字符——并且接着你再学会两种类型, 编组符号 { 和 }。在

第六章, 又出现两种: ~ 和 %。现在你清楚了, 其实有 16 类。这才是问题的全部; 不会再出现更多类型了。任意字符的类代码可以在任何时候改变, 但是保持一个特殊的方案通常比较明智。

要记住的主要问题是, 每个 TEX 格式都为各自的特殊目的而保留一定的字符。例如, 当你使用 plain TEX 格式(附录 B)时, 就需要知道, 下列 10 个字符

```
\ { } $ & # ^ _ % ~
```

在键入时不能按普通方法使用; 它们的每个都使 TEX 做某些特殊的事情, 在本书的某些地方给出解释。如果你实在需要输出这些字符, plain TEX 提供了下列方法:

`\$` 得到 $, `\%` 得到 %, `\&` 得到 &, `\#` 得到 #, `\_` 得到 _;

符号 `\_` 用在计算机程序的*compound_identifiers*。在数学公式中, 可以使用 `\{` 和 `\}` 来得到 { 和 }, 而且 `\backslash` 得到一个反斜线; 例如,

‘`$\{a \backslash b\}$`’ 得到 ‘$\{a\backslash b\}$’.

还有, `\^` 得到的是一个 circumflex 重音(比如, ‘`\^e`’得到的是‘ê’); 并且 `\~` 得到的是一个波浪重音(比如, ‘`\~n`’得到的是 ‘ñ’)。

▶ **练习7.1**

在下列句子中, 会出现什么讨厌的问题?

```
Procter & Gamble's stock climbed to $2, a 10% gain.
```

▶ **练习7.2**

你能想像一下 plain TEX 的设计者为什么不用‘`\\`’这个控制系列代表反斜线吗?

当 TEX 从文件中读入一行或直接从终端上读入一行时, 它把文本转变成一列“记号”。一个记号可以是 (a) 一个指定类代码的单个字符, 或者是 (b) 一个控制系列。例如, 在 plain TEX 的约定下, 文本‘`{\hskip 36 pt}`’转变为一个 8 个记号的列:

$\{_1$ [hskip] 3_{12} 6_{12} $␣_{10}$ p_{11} t_{11} $\}_2$

这里的下标是如前所述的类代码: 1 是“组开始”, 12 是“其它字符”, 诸如此类。[hskip] 没有赋予下标, 因为它表示一个控制系列记号而不是一个字符记号。注意, 记号列中没有 `\hskip` 后的空格, 因为它所跟的是一个控制词。

如果你想对 TEX 有一个完整的理解, 那么理解记号列表的思想很重要的, 把 TEX 的思想看作一个生物体对掌握 TEX 的思想很方便。文件中输入的每个行只能被“眼睛”和“嘴巴”遇见; 但当文本被吞下后, 被按照记号列表的方式送到“胃”里, 而且处理实际排版的消化过程是完全按照记号来进行的。就胃而言, 输入内容就象记号流一样流出, 好象你的 TEX 文稿被一起键入在一个很长的行中。

还记得 TEX 记号的两个主要部分吧: (1) 一个控制系列被看作一个单个对象, 它不是由一系列符号组成的。因此, 当控制系列用记号代替后, TEX 处理长名字与短的是一样的。还有, 在记号列表中间的空格不能忽略; 因为忽略空格的规则只适用于输入文件, 在那期间, 字符串被变成记号。(2) 一个字符记号只能有一个类代码, 这个指定是永久性的。例如, 如果字符'{'突然被声明为类代码为 12 而不是 1, 已经在 TEX 的记号列表中的字符'$\{_1$'的类代码仍然是 1; 只有新的列表才包含记号'$\{_{12}$'。换句话说, 一旦某个字符从文件中被读出来, 那么它的解释就固定了, 就是它读入时的类代码。控制系列却不同, 因为它们可以在任何时间改变它们的解释。TEX 的消化过程总精确地知道一个字符记号表示什么, 因为类代码出现在记号自己身上; 但是当消化过程遇到一个控制系列记号, 为了确定它的意思, 必须找到控制系列当前的定义。

▶ **练习7.3**
0 到 15 中某些类代码从不出现在字符记号的下标上, 因为它们被 TEX 的嘴吃掉了。例如, 类代码为 0 的字符(转义符)从未出现在记号中。哪些类实际出现在 TEX 的胃中?

有一个叫 INITEX 的程序, 它被用来从最开始来安装 TEX; INITEX 类似于 TEX, 只是它的功能更多。它可以把连字符模式压缩到一个特殊的表中从而使得连字化更快, 并且它可以从'plain.tex'得到类似'plain.fmt'的格式文件。但是运行此类任务时, INITEX 需要额外的空间, 因此, 在 TEX 的安装后的版本中, 一般排版可用的内存比你所预期的要少。

当 INITEX 运行时, 它只认识 TEX 的原始控制系列。所有 256 个字符的类代码开始都是 12, 除了 ⟨return⟩ 的是 5, ⟨space⟩ 的是 10, ⟨null⟩ 的是 9, ⟨delete⟩ 的是 15, 52 个字母 A...Z 和 a...z 的是 11, % 和 \ 的分别是 14 和 0。由此, INITEX 最初不能处理依赖于编组的 TEX 某些原始控制系列; 在类代码为 1 和 2 的字符出现之前, 不能使用 \def 或 \hbox。以 \catcode 开始的附录 B 给出了这些必需的类代码; 比如,

```
\catcode`\{=1
```

把符号 { 的类代码指定为 1。操作 \catcode 类似于我们后面要讨论的许多其它 TEX 的原始控制系列; 通过修改象类代码这样的内部量, 可以使 TEX 的应用相当广泛。

▶ **练习7.4**
假定命令

```
\catcode`\<=1 \catcode`\>=2
```

出现在以'{'开始的组的开头; 这些规定告诉 TEX 把 < 和 > 看作组的分隔符。按照 TEX 的局部性规则, 当组结束时, 字符 < 和 > 的类代码将复原回去。但是组应该以 } 还是 > 来结束?

虽然控制系列被看作单个对象, 但是 TEX 的确提供了一种方法来把它分成一列字符记号: 如果输入 \string\cs, 其中 \cs 是任意控制系列, 那么就得到那个控制系列的名字的一个列表。例如, \string\TeX 就得到 4 个记号: $\backslash_{12}$, T_{12}, e_{12}, X_{12}。 在这个记号列表中的每个字符自动得到类代码 12 ("其它字符"), 包括表示转义符的反斜线。但是, 如果字符空格无缘无故出现在控制系列的名字中, 字符'␣'(空格)的类代码将指定为 10。

反过来, 用命令'\csname⟨tokens⟩\endcsname', 可以把一个字符记号列表变成一个控制系列。出现在 \csname 和 \endcsname 之间的这个指令中的记号可以包括其它控制系列, 只要这些控制系列最后展开成的是字符而不是 TEX 的原始控制系列; 最后的字符可以是任意类, 不必是字母。例如, '\csname TeX\endcsname'本质上和 \TeX 是一样的; 但是'\csname\TeX\endcsname'是不合法的, 因为 \TeX 展开为记号时包含原始控制系列 \kern。 还有, '\csname\string\TeX\endcsname'就得到一个不常见的控制系列'\\TeX', 即, 记号 \\TeX, 而你不能用一般的方法来定义它。

▶ **练习7.5**

用 TEX 做个实验, 看看 string 后面跟一个象 ~ 的活动符会出现什么情况。(活动符的表现象控制系列, 但是前面没有转义符。) 在行中做这个实验的简单方法是什么? 为了得到单个字符记号 $\backslash_{12}$, \string 后面应该跟什么?

▶ **练习7.6**

'\expandafter\string\csname a\string\   b\endcsname'得到什么记号? (在 b 前面有 3 个空格。第二十章给出了 \expandafter 的解释。)

▶ **练习7.7**

当 \csname 用来第一次定义控制系列时, 那个控制系列在重新被定义前, 等价于 \relax。利用这个结果来设计一个宏 \ifundefined#1 , 比如, 使得

\ifundefined{TeX}⟨true text⟩\else⟨false text⟩\fi

它在 \TeX 没有被定义, 或者被 \let 等于 \relax 时展开为 ⟨true text⟩; 其它时候展开为 ⟨false text⟩。

在到现在的例子中, \string 把控制系列转变为以 $\backslash_{12}$ 开头的记号列表。但是这个反斜线记号没有真正连接到 TEX 中; 有一个叫 \escapechar 的参数, 它规定了控制系列输出为文本时所使用的字符。\escapechar 的值正常情况下是反斜线的 TEX 内部代码, 但是如果需要其它约定, 可以改变它。

TEX 有两个类似于 \string 命令的其它两个得到记号的命令。如果你输入 \number⟨number⟩, 那么就得到等于 ⟨number⟩ 的小数; 并且如果你输入 \romannumeral⟨number⟩, 就得到用小写 roman 数字表示的数。例如, '\romannumeral24'得到的是'xxiv', 四个记号得到列表中, 每个的类代码都是 12。当用在显示常数时, \number 操作是多于的(比如, '\number24'得到了 '24'); 但是, 它却去掉了开头的零, 并且还可以用在 TEX 内部寄存器或参数中的数字。例如, '\number-0015'得到了'-15'; 并且如果寄存器 \count5 的值为 316, 那么'\number\count5'得到了 '316'。

配对操作 \uppercase{⟨token list⟩} 和 \lowercase{⟨token list⟩} 处理一个给定的记号列表, 并把它们的所有字符转换到相应的"大写"或"小写"字符。如下: 256 个可能的字符都有两个相伴的值, 叫做 \uccode 和 \lccode; 这些值象 \catcode 一样是可以改变的。 转变到大写意味着用 \uccode 值来代替字符, 否则 \uccode 的值就是零(当不改变时)。转变到小写类似, 使用的是 \lccode。类代码不能改变。当 INITEX 运行时, 所有 \uccode 和 \lccode 的值都是零, 除了字母 a 到 z 的 \uccode 为 A 到 Z 和 A 到 Z 的 \lccode 值为 a 到 z。

TEX 在胃中完成 \uppercase 和 \lowercase 的变换, 但是 \string, \number, \romannumeral 和 \csname 操作在到胃的路上就完成了(就象宏的展开), 这些解释在第二十章。

▶ **练习7.8**
'\uppercase{a\lowercase{bC}}'得到是记号列表是什么?

▶ **练习7.9**
TEX 有一个内部整数参数叫做 \year, 它等于任务开始时当前年的数字。看看怎样用 \year 以及 \romannumeral 和 \uppercase 对所有运行在2003的任务给出后面的版权表示: '© MCMLXXXVI'。

▶ **练习7.10**
定义一个三参数控制系列 \appendroman, 使得 \appendroman#1#2#3 定义一个控制系列 #1, 它展开后就是名字为整数参数为 #3的控制系列 #2 的名字, 这个整数参数用 roman 数字来表示。例如, 假定 \count20 等于 30;那么'\appendroman\a\TeX{\count20}'与'\def\a{\TeXxxx}'有同样的效果。

8. 字符输入

使用 TeX 时会遇到不同的键盘, 但是很少有键盘能提供 256 个不同的字符。还有, 就象我们已经看到的那样, 某些可以从键盘键入的字符被保留为象转义符和编组等特殊目的而使用。但是, 当我们讨论字体时就提出了, 每个字体有 256 个字符。那么怎样才能得到不在键盘上或者已经被占用的字符呢?

一个方法是利用控制系列。例如, 在附录 B 的 plain TeX 格式中, 把 `%` 定义为表示注释的一类特殊符号, 就可以定义控制系列 `\%` 来得到百分号。

为了得到任意字符, 可以键入

`\char`⟨number⟩

其中, ⟨number⟩ 是从 0 到 255 的任意数字(后面要跟一个随意的空格); 就会得到当前字体的相应字符。附录 B 也是这样得到 `\%` 的; 它把'`\%`'定义为'`\char37`'缩写, 因为 37 是百分号的字符代码。

TeX 在内部表示字符的代码是基于"ASCII"的, 即美国标准信息交换码。附录 C 给出了这个代码的整个详细资料, 它指定了某些控制符以及普通字母和标点符号的代码。例如, ⟨space⟩ = 32 和 ⟨return⟩ = 13。有 94 个可见的标准符号, 它们的代码是从 33 到 126。

这里, '`b`'在 ASCII 中的字符代码是 98。所以, 如果键盘的 `b` 键坏了, 那么你可以用下列怪方法键入单词 `bubble`:

```
\char98 u\char98\char98 le
```

(常数'`98`'后面的任意空格将被忽略。 当然, 你需要 `\`, `c`, `h`, `a` 和 `r`这些键来键入'`\char`', 所以它们必须是好的。)

TeX 总是使用附录 C 的标准 ASCII 字符的内部字符代码, 不管读入的文件中的外部代码实际的表现。因此, 即使当计算机通常是用 EBCDIC 还是其它非 ASCII 设置来处理, `b` 在 TeX 中也是 98; TeX 软件已经被设置为把文本文件转换为内部代码, 并且当输入文本文件时再转换回外部代码。设备独立(`dvi`)文件使用的是 TeX 内部代码。因此, TeX 在所有计算机上都给出相同的结果。

象附录 C 中的字符代码表通常以八进制的方式给出代码数字, 即, 以 8 为基数的方法, 其中的数字是 *0*, *1*, *2*, *3*, *4*, *5*, *6* 和 *7*。(本手册的作者喜欢用 italic 数字表示八进制数, 以 typewriter 字体表示十六进制数, 目的是尽可能在排版上提示不同的进制。) 有时候, 也使用十六进制, 在这种情况下, 数字为`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E` 和 `F`。例如, '`b`'的八进制代码是*142*, 十六进制代码是 `62`。在 TeX 中, 如果 ⟨number⟩ 的数字前面有 `'` 表示其为八进制, 如果前面有 `"` 将表示十六进制。因此, `\char'142` 和 `\char"62` 等价于 `\char98`。在八进制中, 合法的字符代码是从 *'0* 到 *'377*; 在十六进制中, 则是从 `"0` 到 `"FF`。

但是, TEX 提供了另外一种方法, 使得你根本不必知道 ASCII! 记号 $`_{12}$ (左引号)后面跟任意字符记号或者跟单个字符命名的任意控制系列时, 表示所跟的字符的 TEX 内部代码。例如, `\char`b` 和 `\char`\b` 也等价于 `\char98`。如果你阅读附录 B, 看看 `\%` 是怎样来定义的, 就会发现定义为

```
\def\%{\char`\%}
```

而不是前面声称的 `\char37`。

▶ **练习8.1**

`\def\%{\char`%}`错在哪里?

本手册前言中提到, 作者要不时地编些谎话。唔, 如果你亲自去核对附录 B, 就会发现,

```
\chardef\%=`\%
```

是 `\%` 的真实定义。因为格式的设计者通常希望把一个特殊字符与一个特殊控制系列联系起来, 所以 TEX 给出了一个命令'`\chardef`⟨control sequence⟩`=`⟨number⟩', 其中的数字在 0 和 255 之间, 它可以有效地替代'`\def`⟨control sequence⟩`{\char`⟨number⟩`}`'。

虽然你可以用 `\char` 来得到当前字体的任意字符, 但是它们不能在控制系列中间使用。例如, 如果你键入

```
\\char98
```

TEX 将把它认为是控制系列 `\\`, 后面跟着 `c`, `h`, `a` 等等, 而不是控制系列 `\b`。

在输入文稿时, 你几乎用不到 `\char`, 因为你可能要用到的字符已经有了相应的控制系列; `\char` 主要是为象附录中的那些格式设计者所准备的。但是, 有一天, 你可能需要一个特殊的字符, 并且可能必须在字体的目录中才能找到它。一旦找到后, 你可以直接通过选定相应的字体在用 `\char` 给出字符代码来调用它。例如, 本手册使用的"危险"标志出现在字体 `manfnt` 中, 字符代码是 127, 并且字体用控制系列 `\manual` 选定。因此, 附录 E 中的宏用'`{\manual\char127}`'来给出这个危险标志。

我们已经看到, ASCII 字符集只包括了 94 个可打印的字符; 但是 TEX 要处理从 0 到 255 的 256 个不同的字符, 它们中的每一个都被指定了第七章中讨论的 16 类中的一个。如果你的键盘有额外的符号, 或者如果它不是标准的 94 键, 安装本地 TEX 系统的人会告诉你键入的内容与 TEX 接受到的代码之间的对应关系。有些人很幸运, 键盘上有'≠', '≤'和'≥'; 可以按照 TEX 使得它认识这些便利的符号, 并且使得数学符号的键入更方便。但是如果你没有这些键, 就可以用控制系列 `\ne`, `\le` 和 `\ge` 来得到它们。

TEX 有一个指向不可见 ASCII 字符的标准方法: 代码 0 可以键入为三个字符的系列 `^^@`, 代码 1 为 `^^A`, 等等诸如此类直到代码 31——`^^_`(见附录 C)。 如果跟着 `^^` 的字符的内部代码是 64 到 127, 那么 TEX 将它们的代码减去 64; 如果代码是从 0 到 63, 那么 TEX 要把它加上 64。因此, 代码 127 可以键入为

^^?, 并且危险标志可以用 {\manual^^?} 来得到。但是, 在使用字符的类代码 127 前, 你必须改变它, 因为这个字符一般是 15 类(无用字符); 比如, 用 \catcode`\^^?=12。符号 ^^ 与 \char 不同, 因为 ^^ 可以象单个字符一样组词; 例如, 不允许使用 \catcode`\char127, 但是 ^^ 符号可以象字母一样在控制词中使用。

在第六章, 盒子溢出的一个信息说明了一个问题, TEX 有时候在输出中使用古怪的 ^^ 规则: 在那个例子中的变元音字符以 ^^? 出现, 并且变音符号以 ^^X 出现, 因为‘¨’和‘¸’出现在 \tenrm 字体的*´177* 和*´30*位置。

还有一个特殊的规则, 其中 ^^ 后跟两个“小写十六进制数字”, 0–9 或 a–f。利用这个规则, 所有 256 个字符可以用一种方法得到, 即从 ^^00 到 ^^ff。字符 127 是 ^^7f。

除了不常见的应用外, 大多数 ^^ 代码并不重要。但是 ^^M 特别值得注意, 因为它的代码是 13, 就是 ASCII 的 ⟨return⟩, 一般在输入文件每行的右边结尾。通过改变 ^^M 的类, 可以发挥特殊的作用, 我们将在后面看到它。

控制代码 ^^I 也有潜在的意义, 因为它是 ASCII 的 ⟨tab⟩。Plain TEX 把 ⟨tab⟩ 看作一个空格。

使用非美国字母的人可以把 TEX 转变为其它所要的标准。例如, 假定你有一个挪威键盘, 包含字母 æ, 假定它是斯堪的纳维亚语字母中代码为 241 的字母。你的个人格式包可以定义 \catcode`æ=11, 这样, 你可以使用象 \særtrykk 这样的控制系列了。用 ^^f1 代替字符 241 后, 你的 TEX 输入文件就可以被没有你的键盘的美国式 TEX 系统所读。(例如, 文件中使用的控制系列为 \s^^f1rtrykk; 你的美国朋友也可以使用你的格式, 只要设置 \catcode`^^f1=11 即可。) 当然, 你也可能排列字体使得 TEX 的字符 241 输出为 æ; 并且你应当改变 TEX 连字算法, 使得它能得到正常的挪威语连字符。要点是, 这些变化不是非常困难; TEX 的任何设计都不只限于美国字母。使用的语言不同, 精细地调节后就得到精美的输出。

欧洲语言也可以用一个有限的字符集来得到。例如, 再考虑挪威语, 但是假定你所用的是没有字符 æ 的键盘。你可以改写你的字体度量文件, 使得 TEX 把ae, o/, aa, AE, O/ 和 AA 解释为分别生成 æ, ø, å, Æ, Ø 和 Å 的连字; 并且把字符 å 和 Å 放在字体的 128 和 129 位置上。通过设定 \catcode`/=11, 你将可以在控制系列中象‘\ho/yre’那样使用连字 o/。(当相邻的字符 aa, oe 和 o/ 不是连字时,你的打字员必须能清楚地辨认出; 还有, ‘\/’现在应该是一个控制词而不是一个控制符号。

本章剩下的部分要讲 TEX 的读入规则, 它规定了从文本到记号的转换。例如, TEX 忽略掉控制词后面的空格, 这个结果来自下列一系列的规则, 这些规则意味着控制词后面的空格永远不能变成空格记号。规则将会按照你所期望的方法去实行, 所以你可能不希望麻烦地了解它们; 但是当你与计算机交流时, 看看计算机在认为它正在做什么是有好处的, 这里给你一个机会。

TEX 的输入是一系列“行”。只要 TEX 从文件中或者你在终端所直接输入的一行文本中读入一行文本, 计算机的读入器就处在三个所谓状态的一个:

状态 N	新行;
状态 M	行中间;
状态 S	跳过空格。

在每行的开头, 它处在状态 N; 但是大部分时间它处在状态 M, 在读入控制词或一个空格后, 它处在状态 S。顺便说一下, "状态"与我们以后讨论的"模式"不同; 当前的*状态*是指遇见新文本的字符时 TeX 的眼睛和嘴, 但是当前*模式*是指 TeX 的胃消化时的环境。当把字符转换为记号时, TeX 所做的大部分事情与当前状态无关, 但是当发现空格或行尾字符时(第 10 和 5 类), 就与当前状态有关。

TeX 删除任何出现在行尾的 ⟨space⟩ 字符(代码为 32)。于是, 它在行尾插入有关 ⟨return⟩ 字符(代码为 13), 但是在修复错误时用'`I`'插入的行尾除外。注意, ⟨return⟩ 被看作是实际的字符, 从而是行的一部分; 通过改变它的类你可以得到特殊的效果。

如果 TeX 在任意状态得到一个转义符(第 0 类), 它就如下搜索这个控制系列的名字。(a) 如果在行中没有更多的字符, 那么名字为空的(类似 `\csname\endcsname`)。其次, (b) 如果下一个字符的类不是 11(字母), 那么名字由单个符号组成。最后, (c) 名字由当前字符到最后一个非字母字符前或者到行尾的所有字母组成。这个名字变成一个控制系列记号。在 (c) 或 (b) 的情况下, TeX 进入处理第 10 类的字符的状态 S; 否则 TeX 进入状态 M。

如果 TeX 在任意状态遇见上标字符(第 7 类), 并且此字符所跟的还是另一个一样的字符, 另外这两个相同的字符后面跟一个代码 $c < 128$ 的字符, 那么它们被去掉, 并且从代码 c 中加上或减去 64。(这样, 正如前面讨论的那样, `^^A` 就用代码为 1 的一个单个字符来代替, 等等。) 但是, 如果两个上标字符后面跟的是小写十六进制数字 `0123456789abcdef`, 那么这个四字符的序列被所给十六进制代码的单个字符代替。在上面叙述的搜索控制系列名字的过程的 (b) 和 (c) 中, 如果遇到这样的三字符或四字符, 也同样进行替换。替换完毕后, TeX 重新开始, 就象新字符始终在那里一样。如果上标字符不是这种三字符或四字符的第一个字符, 那么使用下列规则来处理。

如果 TeX 遇见第 1, 2, 3, 4, 6, 8, 11, 12, 或 13 类的一个字符, 或者第 7 类的一个字符且它不是象刚才叙述的特殊序列的开头, 那么它给字符一个类代码, 并且进入状态 M。这是正常的情况; 几乎每个非空白的字符都用这种规则来处理。

如果 TeX 遇见行尾字符(第 5 类), 那么它就放弃本行剩下的所有的内容。于是, 如果 TeX 处在状态 N(新行), 那么行尾字符转换到控制系列记号'`par`'(段结束); 如果 TeX 处在状态 M(行中间), 那么行尾字符转换为第 10 类的字符代码为 32('␣') 的记号(空格); 如果 TeX 处在状态 S(跳过空格), 那么行尾字符就忽略掉。

如果 TeX 遇见要忽略的字符(第 9 类), 那么它直接跳过那个字符, 就象它不在那里一样, 并且保持同样的状态。

如果 TeX 遇见第 10 类的字符(空格), 所得结果与当前状态有关。如果 TeX 处在状态 N 或 S, 字符被直接跳过, 并且 TeX 保持当前状态。否则 TeX 处在状态 M; 字符被转换为第 10 类的记号, 其字符代码为 32, 并且 TeX 进入状态 S。空格记号中的字符代码总是 32。

如果 TeX 遇见注释符(第 14 类), 它就不再读入当前行剩下的内容。

最后, 如果 TeX 遇见无用符(第 15 类), 它跳过此字符, 打印出错误信息, 并且仍保持当前状态。

如果 TeX 在当前行没有要读入的内容了, 那么它转到下一行, 并且进入状态 N。但是, 如果被 `\input` 的文件给出了 `\endinput`, 或者 `\input` 文件结束了, 那么 TeX 返回给出 `\input` 命令的后面。(关于 `\input` 和 `\endinput` 的更详细讨论见第 20 章。)

▶ **练习8.2**
回答下面的简短问题, 看看你对 TeX 的读入规则的理解程度: (a) 第 5 和第 14 类的不同在什么地方? (b) 第 3 和第 4 类的不同在什么地方? (c) 第 11 和第 12 类的不同在什么地方? (d) 活动符后面的空格要忽略掉吗? (e) 当一行以注释符 `%` 结尾时, 在下一行开头的空格被忽略了吗? (f) 一个可以忽略的字符能出现在控制系列的名字中间吗?

▶ **练习8.3**
再次看看出现在有关 `\vship` 的错误信息。当 TeX 报告说 `\vship` 是一个未定义的控制系列时, 它输出了两行上下文, 表明它正在读入文件 `story` 的第二行中间部分。在遇见错误的时候, TeX 处于什么状态? 下一个要读入的是什么字符?

▶ **练习8.4**
给定 plain TeX 的格式的类代码后, 从输入行‘ `$x^2$~  \TeX  ^^62^^6`’得到什么样的记号?

▶ **练习8.5**
想想正好有三行的一个输入文件; 第一行是‘`Hi!`’, 而剩下的两行完全是空的。当 TeX 读入这个文件时, 按照 plain TeX 的类代码, 将得到什么记号?

▶ **练习8.6**
假定 plain TeX 的类代码有效, 字符 `^^A`, `^^B`, `^^C`, `^^M` 分别属于第 0, 7, 10 和 11 类除外。输入行‘`^^B^^BM^^A^^B^^C^^M^^@\M␣`’(相当可笑吧)会得到什么记号? (记住, 此行后面跟着 ⟨return⟩, 即 `^^M`; 并且记住 `^^@` 表示的是 ⟨null⟩ 字符, 当运行 `INITEX` 时它是第 9 类。

在每行结尾插入的特殊字符不必是 ⟨return⟩; TeX 实际上插入的是一个叫 `\endlinechar` 的整数参数的当前值, 它一般等于 13, 但是它可以象其它参数那样可以被改变。 如果 `\endlinechar` 是负数或者大于 255, 那么不添加字符, 结果就象每行都以 `%`(即注释符)结尾。

因为可以改变类代码, 所以 TeX 可以在同一行内使用几个不同类的同一字符。例如, 附录 D 和 E 给出了控制“逐字(verbatim)”处理文本的几种方法, 使得作者可以无大碍地编写本手册。(试着想像排版一个 TeX 手册; 反斜线和其它特殊字符需要在正常类和第 12 类之间换来换去!) 需要注意正确适时地运用, 但是你可以通过巧妙地改变类来得到各种各样的 TeX 结果。另一方面, 最好不要频繁改变类代码, 因为只要它们变成记号, 那么就不能再改变它们的类代码了。例如, 当一个宏的变量第一次被搜索到时, 它们就被放置在记号列中, 因此, 它们的类只要一次固定将从此全部都固定。为了阻止人们在普通情况下太多地使用 `\catcode` 的变化, 作者有意把类代码保留为数值格式而不是容易记忆的格式。

▶ **练习8.7**
附录 B 用 `\lq` 和 `\rq` 来定义 ‘ 和 ’ (分别是左右单引号)。解释一下, 为什么下列定义并不是很好。

9. TEX 的 Roman 字体

当你输入 TEX 文稿时, 你需要知道有哪些可用的符号。附录 B 的 plain TEX 格式使用的基本字体是 Computer Modern 字体, 它可用于多种文档。现在来看看当直接输入文本时 plain TEX 可用干些什么。我们已经遇到过一些有点微妙的问题——例如, 在第二章讨论的破折号和引号, 在第三和第六章出现的一些重音符。本章的目的是通过汇总所有的情况, 得到更系统的总结。

首先以正常的 roman 字体(`\rm` 或 `\tenrm`)的规则开始; 如果你不给出字体, 那么 plain TEX 将在所有情况下使用本字体。你需要的大部分普通符号都容易得到, 并且你可以正常地输入它们: 对于

字母 `A` 到 `Z` 和 `a` 到 `z`

数字 `0` to `9`

一般的标点符号 `: ; ! ? ( ) [ ] ‘ ’ - * / . , @`

没有什么特别之处, 但是 TEX 将把下列组合字看作某些组合:

`ff` 得到 ﬀ;	`ffi` 得到 ﬃ;	`‘‘` 得到 “;	`!‘` 得到 ¡;
`fi` 得到 ﬁ;	`ffl` 得到 ﬄ;	`’’` 得到 ”;	`?‘` 得到 ¿.
`fl` 得到 ﬂ;	`--` 得到 –;	`---` 得到 —;	

还可以输入 `+` 和 `=` 来得到相应的符号 $+$ 和 $=$; 但是只在数学模式中使用它们更好一些, 即用两个 `$` 把它们夹起来, 因为这会告诉 TEX 要插入适当的数学间距。数学模式在后面讨论; 现在, 只需要记住公式和文本要隔开。非数学的连字符和斜线应该在数学环境外输入为‘`-`’和‘`/`’, 而减号和除号应该是 `$` 号之间输入‘`-`’和‘`/`’。

前一段讨论了标准 ASCII 的 94 个可见字符中的 80 个; 因此, 你的键盘可能还包含至少 14 个符号, 并且对剩下的符号你应当小心, 因为它们是特殊符号。它们中的四个被 plain TEX 预先占用了; 如果你的文稿需要下列符号:

`$    #    %    &`

那么你应该分别这样输入它们:

`\$    \#    \%    \&`

Plain TEX 还保留了六个符号:

`\    {    }    ^    _    ~`

但是你可能并不在乎失去它们, 因为一般的文档不需要它们。在数学模式中, 通过控制系列可以使用括号和反斜线。

在标准 ASCII 集中, 还有四个剩下的特殊符号:

```
"     |     <     >
```

当你排版文本时, 并不真正要用它们。(双引号可以用 `` 或 '' 来代替; 竖线和关系号只在数学模式下用到。)

英文的学术出版物通常要用到其它语言, 所以 plain TEX 可以排版最常用的重音符:

输入	得到	
`` \`o ``	ò	(grave accent)
`\'o`	ó	(acute accent)
`\^o`	ô	(circumflex or “hat”)
`\"o`	ö	(umlaut or dieresis)
`\~o`	õ	(tilde or “squiggle”)
`\=o`	ō	(macron or “bar”)
`\.o`	ȯ	(dot accent)
`\u o`	ŏ	(breve accent)
`\v o`	ǒ	(háček or “check”)
`\H o`	ő	(long Hungarian umlaut)
`\t oo`	o͡o	(tie-after accent)

在字体内, 这样的重音符在设计上是正好出现在字母‘o’的高度; 但是可以在任意字母上使用它们, 并且字母变高时 TEX 会自动升高重音符。注意, 在后四种情况, 分开控制系列和后面字母的空格是必须的。但是, 为了避免在单词中间出现空格, 应该输入‘`\H{o}`’。

Plain TEX 还给出了出现在下面的三个重音符:

输入	得到	
`\c o`	o̧	(cedilla accent)
`\d o`	ọ	(dot-under accent)
`\d {\hbox{国}}`	国̣	(dot-under accent)
`\b o`	o̱	(bar-under accent)

还有几个特殊的字母:

输入	得到	
`\oe,\OE`	œ, Œ	(French ligature OE)
`\ae,\AE`	æ, Æ	(Latin and Scandinavian ligature AE)

`\aa,\AA`	å, Å	(Scandinavian A-with-circle)
`\o,\O`	ø, Ø	(Scandinavian O-with-slash)
`\l,\L`	ł, Ł	(Polish suppressed-L)
`\ss`	ß	(German “es-zet” or sharp S)

`rm` 字体还包含无点字母‘ı’和‘ȷ’, 你可以用‘`\i`’和‘`\j`’来得到它们。这是有用的, 因为当‘i’和‘j’得到重音符时应该去掉上面的点。例如, 得到‘mīnŭs’的正确方法是输入‘`m\=\i n\u us`’或‘`m\={\i}n \u{u}s`’。

这就完成了我们对 `\rm` 字体的总结。确切地说, 同样的约定对 `\bf`,`\sl` 和 `\it` 也可以, 所以当使用不同字体时, 不必用不同的方法。例如, `\bf\"o` 得到的是 **ö**, `\it\&` 得到的是 *&*。奇妙吧?

但是, `\tt` 稍有不同。当你使用 typewriter 字体时, 乐意见到 `ff`, `fi` 等等不被当做连字来处理; 也不会从破折号和引号得到组合字。这很好, 因为当你模仿打字机时, 普通破折号和普通双引号比较合适。大部分重音符也可以使用。但是 `\H`, `\.`, `\l` 和 `\L` 不能使用——typewriter 字体在这些地方对应于其它符号。当前, 你忽然可以输入 `"`, `|`, `<` 和 `>` 了; 见附录 F。在 `\tt` 中, 所有的字母, 空格和其它符号的宽度相同。

▸ **练习9.1**

用非正常的方式输入‘naïve’。

▸ **练习9.2**

列出包含重音字母的一些英语单词。

▸ **练习9.3**

怎样输入‘Æsop’s Œuvres en français’?

▸ **练习9.4**

为了得到下列句子, 看看应该输入什么? *Commentarii Academiæ scientiarum imperialis petropolitanæ* is now *Akademii͡a Nauk SSSR, Doklady*.

▸ **练习9.5**

怎样得到下列名字? Ernesto Cesàro, Pál Erdős, Øystein Ore, Stanisław Świerczkowski, Sergeĭ I͡ur’ev, Muḥammad ibn Mûsâ al-Khwârizmî?

▸ **练习9.6**

设计一种方法, 用 typewriter 字体排版 `Pál Erdős`。 Devise a way to typeset in typewriter type.

不管你使用 `\rm`, `\sl`, `\bf`, `\it` 或 `\tt`, 下列符号都是一样的:

输入	得到	
`\dag`	†	(dagger or obelisk)

`\ddag`	‡	(double dagger or diesis)
`\S`	§	(section number sign)
`\P`	¶	(paragraph sign or pilcrow)

(它们仅仅以一种字体出现, 因为 plain TeX 是从数学符号中得到它们的。数学上需要许多其它符号; 我们将在后面讨论它们。附录 B 还有几个非数学符号。)

▸ **练习9.7**

在 plain TeX 的 italic 字体中, '\$'得到的是'*£*'。这是得到英镑符号的一种方法, 但是你可能要一个 italic 美元符号。你能想出怎样排版书名 *Europe on \$15.00 a day* 吗?

附录 B 表明, plain TeX 用原始控制系列 `\accent` 来处理大多数重音符。例如, `\'#1` 等价于 `{\accent19 #1}`, 这里的 `#1` 是要加上重音的变量。一般规则是, `\accent`⟨number⟩ 把重音加在下一个字符上; ⟨number⟩ 给出了当前字体中本重音出现的位置。假定了重音正好放在当前字体中高度为 x 的高度的字符上; 更高或更矮的字符会使重音符升高或降低, 而且还要考虑重音符和字符的字体的倾斜问题。最后输出字符的宽度是字符的宽度, 与重音符的宽度无关。象字体变换这样不依赖于模式的命令可以在重音符和字符之间出现, 但是编组不能交叉。如果没有合适的字符, 那么出现的是重音符自己, 相当于 `\char`⟨number⟩, 而不是 `\accent`⟨number⟩。例如, `\'{}` 得到的是 ´。

▸ **练习9.8**

想想为什么 plain TeX 把 `\'#1` 定义为'`{\accent19 #1}`', 而不是直接用 `\'` 来代替'`\accent19 `'? (为什么有一个额外的括号? 为什么有变量 `#1`?)

重要的是记住, 我们所讨论的重音和特殊字符的约定不是 TeX 自带的; 它们只属于 plain TeX 格式, 而本格式使用的是 Computer Modern 字体。当涉及其它字体时, 相应的约定差别很大; 格式设计者应该在它们的特殊系统内提供得到重音和特殊字符的方法。Plain TeX 对不经常使用的重音也处理得很好, 但是本章的约定无法推荐给其它语言的 TeX 的大范围应用。例如, 法语中设计得比较好的 TeX 字体是把重音当做组合字, 因此可以不用反斜线就得到 `e'crire de cette manie`re nai"ve en franc/ais`。(见第八章关于挪威语的讨论。)

10. 尺寸

有时候, 需要告诉 TEX 产生多大的间距, 或者生成一条多长的直线。例如, 第六章简短的 stroy 中, 使用了命令‘`\vskip .5cm`’来在垂直方向跳过半厘米, 并且用‘`\hsize=4in`’给出了宽度为 4 英寸的水平栏。现在, 应该讨论一下把这些尺寸要求告诉 TEX 的各种方法了。

在使用英语的国家, “points”和“picas”是打印机和排版界的传统的度量单位, 所有 TEX 认识 points 和 picas。TEX 还认识英寸制和米制, 以及欧洲大陆使用的 points 和 picas。每个度量单位用两个字母的缩写表示如下:

`pt` point (本手册的基线之间距离是 12 pt)
`pc` pica (1 pc = 12 pt)
`in` inch (1 in = 72.27 pt)
`bp` big point (72 bp = 1 in)
`cm` centimeter (2.54 cm = 1 in)
`mm` millimeter (10 mm = 1 cm)
`dd` didot point (1157 dd = 1238 pt)
`cc` cicero (1 cc = 12 dd)
`sp` scaled point (65536 sp = 1 pt)

利用列在这里的转换因子作为精确的比例, TEX 严格按照米制系统输出。

▶ **练习10.1**
在 254 厘米中有多少 points?

当你要告诉 TEX 某些物理尺寸时, 如下输入:

⟨optional sign⟩⟨number⟩⟨unit of measure⟩

或者

⟨optional sign⟩⟨digit string⟩`.`⟨digit string⟩⟨unit of measure⟩

其中, ⟨optional sign⟩ 可以是‘`+`’或‘`-`’, 也可以什么也没有, ⟨digit string⟩ 由零或更多的连续的小数组成。‘`.`’也可以是‘`,`’。例如, 下面是六个典型的尺寸:

```
3 in              29 pc
-.013837in        + 42,1 dd
0.mm              123456789sp
```

正号是多余的, 多少有些人喜欢偶尔画蛇添足。在正负号, 数字和测量单位前面的空格可有可无, 也可以在尺寸后面放一个空格; 但是在数字的数中间, 或者在测量单位的字母之间不能有空格。

▶ **练习10.2**

把上面的六个“典型的尺寸”从小到大排序。

▶ **练习10.3**

下面的三个尺寸有两个是符合 TEX 的规则的。它们是哪两个？意思各是什么？

```
'.77pt
"Ccc
-,sp
```

下面的“标尺”已经被 TEX 画出来了，这样你可以通过互相比较它们来体会一下它们的差别。如果 TEX 输出结束后在打印过程中没有发生变形，这些标尺是非常精确的。

4 in

300 pt

300 dd

10 cm

▶ **练习10.4**

(在学会盒子和粘连以及看完第二十一章后再做。) 看看怎样用 TEX 画出 10 cm 的这样的标尺。

在 TEX 内部把所有尺寸表示为一个叫做 sp 的小单位的整数倍。因为可见光的波长近似等于 100 sp，几个 sp 的误差眼睛是看不出来的。但是，TEX 非常仔细地进行计算，使得在不同计算机上得到的结果相同。对同一文档，TEX 的不同运行环境却能得到同样的断行和分页，因为它所应用的整数算法是相同的。

在这里，给定单位是为了转换为 sp 时在各种计算机上的效率都很高。为此，TEX 的“pt”比官方打印机的 point 略大，在 1886 年，American Typefounders Association 把point 精确地定义为 .013837 in[参见 National Bureau of Standards Circular 570 (1956)]。实际上，一个传统的 point 精确地等于 .99999999 pt，因此“误差”最大在 10^8 分之几。这个差比 1959 年英寸从先前的 (1/0.3937) cm 缩到 2.54 cm 时减小的量小两个量级；所以不必在意这个差别。新定义 72.27 pt = 1 in 不仅好计算，而且容易记住。

TEX 不能处理绝对值大于等于2^{30} sp的尺寸。换句话说，最大的可能的尺寸比 16384 pt 略小。这大概是 18.892 feet(5.7583 米)，所以它是不会束缚你的。

在类似本书这样的编程的手册中，对象 ⟨number⟩，⟨optional sign⟩ 和 ⟨digit string⟩ 这些指令使用“角括号”表示缩写是很方便的。因此，我们将用 ⟨dimen⟩ 表示 TEX 允许的尺寸。例如，

\hsize=⟨dimen⟩

就是 TEX 设置的定义栏宽度的一般方法。意思就是, ⟨dimen⟩ 可以用任何象'`4in`'这样符合 TEX 语法规则的尺寸的量来代替; 角括号中的缩写把这样的语法规律叙述得更清楚。

当尺寸为零时, 你必须给出一个测量单位, 即使它没什么关系。不能只使用'`0`'; 应该使用'`0pt`', '`0in`'或其它东西。

在本手册中你所看到的是 10-point 的大小, 但是可能你会经常遇到要使用更大的字体的时候。Plain TEX 通过放大输出来很容易地实现它。如果你在文稿的开头规定

```
\magnification=1200
```

那么所有的内容都被放大 20%; 即, 正常尺寸的 1.2 倍。类似地, '`\magnification=2000`'把所有内容放大一倍; 这实际上把每个字符的面积变成四倍, 因为高度和宽度都加倍了。为了按照因子 f 来放大文档, 你可以规定 `\magnification=`⟨number⟩, 其中 ⟨number⟩ 是 1000 乘以 f。这个指令必须在第一页输出完成之前给出。在同一文档不能使用两个不同的放大率。

放大功能有一个明显的好处: 当你在校对时, 眼睛不会感到太累; 可以简单地得到演讲用的透明胶片; 还可以弥补低分辨率打印机对输出图像的造成的损失。反过来, 为了把某些书变成口袋大小, 你可能要用'`\magnification=500`'。但是有个条件: 如果你的打印设备凑巧没有你所要放大的字体, 那么就不能使用放大功能。只要你按照 `\magstep0`, `1`, `2`, `3`, 甚至是 `4` 和 `5`(见第四章)来放大, 那么可能大部分安装好的 TEX 都能打印所有 plain TEX 的字体; 但是使用大字体可能花费比较大, 因为通常需要许多内存空间来储存字形。

▶ **练习10.5**
试试按照正常尺寸的 1.2, 1.44 和 1.728 倍来输出第六章的 story。应该怎样做才可以?

当你规定了 `\magnification=2000` 后, 象'`\vskip.5cm`'这样的结果实际上在最后的文档中是跳过了 1.0 cm 的空白。如果你要给出一个不变的尺寸, TEX 允许在 `pt`, `pc`, `in`, `bp`, `cm`, `mm`, `dd`, `cc` 和 `sp` 紧前面加上'`true`'。这是不被放大的尺寸, 使得后面的放大被取消。例如, 如果你在前面规定了'`\magnification=2000`', 那么'`\vskip.5truecm`'等价于'`\vskip.25cm`'。Plain TEX 在 `\magnification` 命令自己中使用了这个特性: 附录 B 就在新的放大指令起作用后, 包括了指令

```
\hsize = 6.5 true in
```

它调整行的宽度, 使得最后输出时, 在每页的内容都是 $6\frac{1}{2}$ 英寸宽, 而与放大因子无关。设定纸的宽度为 $8\frac{1}{2}$ 英寸, 那么在左右各有 1 英寸的边界。

如果你没有使用'`true`'尺寸, 那么 TEX 的内部运算不会因为放大的出现或不存在而受到影响; 断行和分页是一样的, 并且 `dvi` 文件只改变两个地方。TEX 将直接告诉打印机你想要的放大率, 当读入 `dvi` 文件时, 打印机将给出实际的放大输出。

▶ **练习10.6**
第四章提到, 通过'`at`'不同大小载入字体, 不同放大率的字体可以在同一文档中使用。当你给出下列命

令时, 看看使用的是什么字体:

```
\magnification=\magstep1
\font\first=cmr10 scaled\magstep1
\font\second=cmr10 at 12truept
```

放大功能实际上是由 TEX 的原始控制系列 \mag 所控制, 它是一个整数参数, 应当是正数且最大为 32768。\mag 的值在三种情况下被用到: (1) 在第一页被输出到 dvi 文件之前; (2) 当计算一个 true 尺寸时; (3) 当 dvi 文件结束时。另外, 有些 TEX 的输出不是 dvi 文件; 这时在情况(2)下并且当输出到每页时要用到 \mag。因为每个文档只有一个放大率, 所以一旦 \mag 的值第一次被用到后, 就不能改变了。

TEX 还有两个相对而不是绝对的测量单位; 即, 它们与当前的上下文有关:

em 是当前字体的一个"quad"的宽度;
ex 是当前字体的"x-height"。

每个字体都有它自己的 em 和 ex 值。在过去, "em"是'M'的宽度, 但是现在不是了; em 是直接来自字体的任意单位, ex 也是。对 Computer Modern 字体, em 破折号的宽度是一个 em, 每个数字 0 到 9 的宽度是半个 em, 小写'x'的高度是一个 ex; 但是不是对所有字体都这样。Plain TEX 的 \rm 字体(cmr10)为 $1\,\mathrm{em} = 10\,\mathrm{pt}$, $1\,\mathrm{ex} \approx 4.3\,\mathrm{pt}$; \bf 字体(cmbx10)为 $1\,\mathrm{em} = 11.5\,\mathrm{pt}$ 和 $1\,\mathrm{ex} \approx 4.44\,\mathrm{pt}$; \tt 字体(cmtt10)为 $1\,\mathrm{em} = 10.5\,\mathrm{pt}$ 和 $1\,\mathrm{ex} \approx 4.3\,\mathrm{pt}$。它们都是"10-point"字体, 但是 em 和 ex 值却不同。对当前字体的水平距离最好用 em, 对垂直距离最好用 ex。

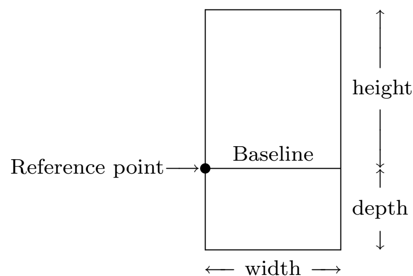
⟨dimen⟩ 也能用 TEX 的内部寄存器或参数。我们在后面将讨论寄存器, ⟨dimen⟩ 的所有内容的完整定义在第二十四章给出。'\hsize'是当前行的宽度, 那么'.5\hsize'是那个量的一半; '2\wd3'表示寄存器 \box3 的两倍; '-\dimen100'是寄存器 \dimen100 的负值。

注意, 在尺寸中, 单位的名称前面没有反斜线。TEX 语言的其它所谓关键词也是如此。关键词可以用大写或大写小写混合起来; 比如, 'Pt'等价于'pt'。这些字母的类代码无关; 例如你可以正在用第 12 类(其它字符)的 p, 正如第二十章讨论的那样所展开的'\the\hsize'生成。只要当关键词出现在某些非常有限的上下文中时, TEX 才给出特殊解释。例如, 只有当出现在 ⟨dimen⟩ 中的数字后面时'pt'才是关键词; 只有当出现在 \font 声明中的字体外部名字后面时, 'at'才是关键词。如果你想知道所有的关键词, 下面就是 TEX 的全部关键词的列表: at, bp, by, cc, cm, dd, depth, em, ex, fil, height, in, l, minus, mm, mu, pc, plus, pt, scaled, sp, spread, to, true, width。(至于在哪些情况下它们被看作关键词, 见附录 I 给出的参考。)

11. 盒子

在制作复杂页面时, TeX 首先把简单的单个字符放在一起生成一个大的单元, 然后再把大的单元放在一起生成更大的单元, 等等。在概念上, 这就是一个繁重的粘连工作。用来描写这样的页面构造的 TeX 术语是*盒子*和*粘连*。

TeX 中的盒子是长方形的二维对象, 有三个相应的尺寸, 叫做*高度*, *宽度*和*深度*。下面是一个标准盒子的样子, 其中给出了它的所谓基准点和基线:



在 TeX 看来, 来自字体非单个字符就是一个盒子; 它是一种最简单的盒子。字体的设计者已经给出了字符的高度, 宽度和深度, 以及它在盒子中的形状; TeX 利用这些尺寸把盒子粘连在一起, 并且最后确定页面中所有字符的基准点的位置。例如, 在 plain TeX 的 `\rm` 字体(`cmr10`)中, 字母‘h’的高度是 6.944 points, 宽度为 5.5555 points, 深度为零; 字母‘g’的高度为 4.3055 points, 宽度为 5 points, 深度为 1.9444 points。虽然 `cmr10` 叫做“10-point”字体, 却只有象圆括号这样的特殊字符的高度加深度正好等于 10 points。实际上你并不需要知道这些尺寸, 但是知道了这些 TeX 处理的内容是有益的; 这样就能更好地理解计算机是怎样处理你的文稿的。

字符形状不需要处在盒子的边界内。例如, 有些字符被用来构建更大的数学符号, 象矩阵的括号就有意突出来一点, 这样就能与其它符号适当地重叠起来。Slanted 字母常常超出盒子的右边界, 就象盒子在顶部向右倾, 在底部向左倾, 而基线保持不动一样。例如, 比较 `cmr10` 和 `cmsl10` 字体(`\rm` 和 `\sl`)中的字母‘g’:

cmr10 cmsl10

g g 放大了 3 倍

在两种情况下, TeX 都认为盒子的宽度是 5 points, 所以两个字母正好采用同样的处理方式。TeX 不考虑形状如何——只有输出设备知道形状。但是因为基线是知道的, 所以尽管 TeX 没什么别的信息, slanted 字母也能放在适当位置上。

实际上, 字体设计者也要告诉 TeX 一个信息, 就所谓的 *italic* *修正*: 对每个字符要规定一个数字, 大概说明字符要伸出其右边界多少, 再加上一点富余量。例如, 在 `cmr10` 中字母‘g’的 italic 修正为 0.1389 pt, 而在 `cmsl10` 中为 0.8565 pt。第四章指出, 如果你在字符紧后面输入了‘`\/`’, 那

么这个修正就要加到正常宽度上。当从 slanted 字体转到 unslanted 字体时记住要用 \/, 特别是在象

```
the so-called {\sl italic correction\/}:
```

的情况, 因为这里没有插入的空格去弥补 slant 的占用。

TEX 处理的另外一种简单的盒子可以称为"黑色盒子", 即象'▮'这样的长方形, 在打印时要完全上色。对这样的盒子, 你可以给出任意高度, 宽度和深度——但是最好别太大, 否则打印墨可能耗尽。(打印时最好使用白色格子而不是黑色的。)

通常这些黑色盒子被定得很细, 使得看起来象水平直线或垂直直线。传统上打印机将其称为"水平标尺"和"垂直标尺", 因此在 TEX 中表示这些黑色盒子的术语为 `\hrule` 和 `\vrule`。即使盒子是方的, 就象'▪'一样, 你也得称其为 `\hrule` 或 `\vrule`。后面我们将更详细讨论标尺盒子的用法。(见第二十一章。)

TEX 所排版的所有页面上的内容都由简单的字符盒子或标尺盒子组成, 在组合时粘连在一起。TEX 用两种方法把盒子粘连在一起, *在水平方向上或者在垂直方向上*。当 TEX 建立了一个盒子的水平列时, 就把它们对齐, 使得其基准点在同一水平行上; 从而相邻字符的基线按部就班地匹配好。类似地, 当 TEX 建立了盒子的垂直列时, 也要对齐它们, 使得其基准点在同一垂直列上。

通过对比计算机的方法和手工排版来看看在幕后 TEX 是怎样工作的。在久经考验的传统方法中, 首先从铅字格中选好需要的字母——大写字母在上面的格中——并且把它们放在"组合盘"中。当完成一行时, 要调整好间距并且把它放在与其它排好的行合并起来的"钢框"内。最后, 通过调整外面的楔子把版锁紧, 即"楔紧"。这与 TEX 所做的大同小异, 只不过称呼不同; 当 TEX 锁好一行后, 就得到一个所谓的"hbox"(水平盒子), 因为行的各个部分是水平连接在一起的。在 TEX 的文稿中, 你可以给出象

```
\hbox{A line of type.}
```

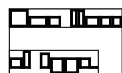
这样的指令; 它告诉计算机把当前字体的相应字母放在盒子中, 并且把它们锁在一个 hbox 中。对 TEX 而言, 字母'A'是一个盒子'□', 字母'p'是一个盒子'▫'。所以所给的指令使 TEX 去制作一个盒子



来表示'A line of type'。排版得到的各个行的 hbox 最后放在一个"vbox"(垂直盒子)中来合并起来。例如, 你可以输入

```
\vbox{\hbox{Two lines}\hbox{of type.}}
```

TEX 将把它转变为

 i.e., Two lines of type.

TEX 方法和老方法的主要差别是, 铅字排版一般是铸件, 使得每个字符的高度和宽度相同; 这使得容易手工对齐。TEX 排版中, 字符高度和深度各不相同, 因为用基线对齐时不会产生影响, 并且因为与高度和深度有关的特殊信息有利于对重音和数学符号的定位。

当然, TEX 排版和手工排版的另一个重要差别是, TEX 自动生成分开的行; 你不必插入 \hbox 和 \vbox 指令, 除非要完全控制每个字母的位置。另一方面, 如果的确使用了 \hbox 和 \vbox, 那么你可以完成 Ben Franklin 在他的印刷厂中所做的任何工作。你正在放弃的仅仅是龙飞凤舞的墨水字母; 要得到这样的效果就需要新的字体。(当然你还失去了触觉和嗅觉, 以及亲自操刀的快感。TEX 将从不完全替代老的方法。)

在 TEX 看来, 你正在看的这页本身就是一个盒子: 它是从表示每行文本的更小的盒子的垂直列制作的相当大的盒子。依次地, 每行文本是表示各个字符的盒子的水平列制作的盒子。在包括数学公式和/或复杂表格的更复杂的情形, 可以盒子套盒子套盒子 . . . 到任意多层。但是, 即使是这样复杂的情形也是按照简单的方法从粘连在一起盒子的水平或垂直列得到; 每次你和 TEX 所必须关心的所有东西就是一个盒子列。实际上, 当排版一个简单文本时, 根本不必考虑盒子, 因为 TEX 会自动负责把字符变成单词, 把单词变成行, 把行变成页的工作。只有当你要做一些非普通的排版——比如把标题居中——时, 才需要盒子的知识

从 TEX 消化过程来看, 文稿按照一系列记号进去, 并且记号被转换为一系列盒子。每个输入的记号本质上是一个指令或者一批指令; 例如, 记号'A_{11}'的正常意思是, "用当前字体, 把字母 A 的字符盒子放在当前盒子的最后"; 记号'[vskip]'的正常意思是, "在当前盒子中垂直跳过下一个记号给出的 ⟨dimen⟩"。

盒子的高度, 宽度或深度可能是负值, 这时它是有些难画出的"隐性盒子"。TEX 不因负尺寸而出现问题; 仅仅是按照通常的算法进行处理。例如, 两个相邻盒子的组合宽度是它们的宽度之和, 而不管它们的宽度是不是正值。字体设计者可以把一个字符的宽度声明为负值, 这时, 字符实际上与一个退格一样。(对从右到左的语言使用本方法来处理, 但是也有一定限度, 因为 TEX 的断行算法是基于单词的宽度不是负值的假定的。)

在水平列中, TEX 可以升高或降低各个盒子; 这样的调整照顾到了数学中的上下标, 以及重音的高度和其它几种情况。例如, 这里是设定包含 TEX 标识的盒子的方法, 把它放在了 TEX 的内部寄存器 \box0 中:

```
\setbox0=\hbox{T\kern-.1667em\lower.5ex\hbox{E}\kern-.125em X}
```

这里的'\kern-.1667em'就是插入当前字体的一个 $-.1667em$ 的空格, 即, 退回一点; 还有, '\lower.5ex'就是盒子 \hbox{E} 被降低当前的 x 的高度的一半, 因此此盒子相对于其它盒子就偏移了。不用'\lower.5ex', 也可以用'\raise-.5ex'。第十二章和第二十一章详细讨论了怎样得到特殊效果的盒子; 本章的目标仅仅是牛刀小试。

如果要问盒子寄存器的内容, TEX 将给出。例如, 如果在如上对 TEX 的标识设置完毕 \box0 后, 你输入'\showbox0', 那么 log 文件将包含下列叽哩咕噜的东西:

```
\hbox(6.83331+2.15277)x18.6108
.\tenrm T
.\kern -1.66702
.\hbox(6.83331+0.0)x6.80557, shifted 2.15277
..\tenrm E
.\kern -1.25
.\tenrm X
```

第一行表示 \box0 是一个 hbox, 其高度, 深度和宽度分别为 6.83331 pt, 2.15277 pt 和 18.6108 pt。后面的行以'.'开头表明它们是盒子中的内容。在这个特殊盒子中, 首先是 \tenrm 字体的字母 T; 接着是一个 kern。下一项是只包含字母 E 的 hbox; 这个盒子的高度, 深度和宽度与 E 一样, 并且它向下移动了 2.15277 pt(因此得到了更大的盒子深度)。

▶ **练习11.1**
为什么在'..\tenrm E'这行开头有两个点?

这样的盒子内容表示在第十二和第十七章中将进一步讨论。当你试图精确地解决 TEX 所处理的情况时, 它们主要用于查找错误。把它们在本章进行讨论的主要原因是为了领会一下 TEX 是怎样在读入时表示盒子的。计算机程序并不真正把盒子弄来弄去; 它处理的是盒子的表示列表。

▶ **练习11.2**
通过运行 TEX, 看看它实际上怎样来处理字符的 italic 修正的: 在盒子中修正怎样表示?

▶ **练习11.3**
"反向"的 TEX 标识——即 T^EX——由下列得到

```
\setbox1=\hbox{T\kern+.1667em\raise.5ex\hbox{E}\kern+.125em X}
```

现在, \showbox1 会出现什么?(先猜猜看, 并直接运行。)

▶ **练习11.4**
想想为什么 TEX 的作者不把盒子在水平和垂直上设定得更对称一些, 只需允许基准点在边界内部而不是坚持把它放在每个盒子的左边界上?

▶ **练习11.5**
构造一个宏 \demobox, 用来得到如此的手册, 使得作者可以输入'\demobox{Tough exercise.}'来排版出'[Tough exercise.]'。

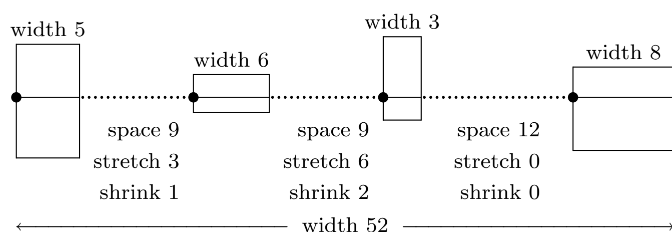
▶ **练习11.6**
构造一个宏 \frac, 使得'\frac1/2'得到的是'½'。

12. 粘连

但是还有比盒子更多的东西: 称之为*粘连*的魔术泥巴, TEX 就是用它把盒子粘在一起的。例如, 本手册的文本行之间有小空隙; 它正好使得同一段中的相邻行的基线精确地分开 12 points。并且在单词之间也有空隙; 这样的粘连可以伸缩, 使得每页的右边界看起来是对齐的。

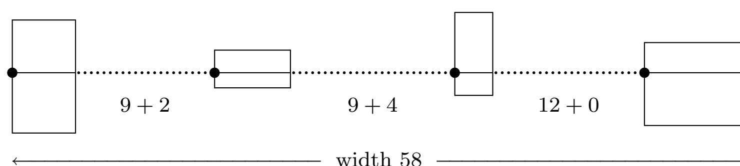
当 TEX 把更小盒子的水平或垂直列组合成大盒子时, 通常在小盒子之间都有粘连。粘连有三种属性, 即*正常间距*, *伸长能力*和*收缩能力*。

为了理解其工作原理, 看看下面在一个水平列中被三个粘连团分开的四个盒子的例子:



第一个粘连单元是 9 个单位空格, 伸长 3 个, 收缩 1 个; 下一个也是 9 个单位空格, 但是伸长 6 个, 收缩 2 个; 最后一个是 12 个单位空格, 但是不能伸缩, 所以不管怎样它也要保持 12 个单位空格。

不考虑粘连的伸缩时, 本例中的盒子和粘连的总宽度是 $5 + 9 + 6 + 9 + 3 + 12 + 8 = 52$ 个单位。它称为水平列的*自然宽度*; 是把盒子粘在一起的最好结果。但是, 假定要求 TEX 把盒子变成宽度为 58 个单位; 那么粘连必须伸长 6 个单位。好, 目前的伸长能力是 $3 + 6 + 0 = 9$ 个单位, 所以为了得到所需要的 6 个单位, TEX 用 6/9 乘以每个可伸长的单位。第一个粘连团变成 $9 + (6/9) \times 3 = 11$ 个单位的宽度, 第二个变成 $9 + (6/9) \times 6 = 13$, 最后一个保持 12 个单位不变, 我们将得到了如下这样的符合要求的盒子:



另一方面, 如果要求 TEX 使给定列的盒子宽度为 51 个单位, 就必须使粘连整体收缩 1 个单位。目前的收缩能力是 3 个单位, 所以第一个收缩 1/3, 第二个收缩 2/3。

当从水平或垂直列制作盒子时, 确定粘连的厚度的过程称为*设定粘连*。一旦粘连被设定, 它就变成固定的了; 就不能在伸缩了, 并且得到的盒子本质上不能再分解开。

粘连的收缩不得超过其收缩能力。例如, 在我们的图示中的第一个粘连团的宽度不允许小于 8 个单位, TEX 也不能使给定水平列的总宽度小于 49 个单位。但是粘连允许任意伸长, 只要伸长量是正值。

▸ **练习12.1**

如果图示中的水平列的盒子的宽度为 100 个单位, 粘连团是多宽?

一旦你理解了 TEX 的粘连的概念, 就会认为名不符实; 真正的粘连不能用这样的方法伸缩, 也不能在粘接的盒子之间提供很多间隙。象"弹簧"这样的词才能更贴近原意, 因为弹簧有一个自然宽度, 并且在拉压下不同弹簧的伸缩比例不同。但是只要作者提出改变 TEX 的术语, 许多人就说它们喜欢词"粘连", 尽管有点不合适; 所以原来的名字还是保留下来了。



超出粘连规定的伸长能力的伸长对 TEX 有点多余; 然而, 通过下列规则, 你可以确定粘连的各项内容: (a) 自然粘连空格应该是看起来最好的空格的量。(b) 粘连的伸长应该是版面变糟之前可以加在自然空格上的最大空格量。(c) 粘连的收缩应该是版面变糟之前可以从自然空格中减去的最大空格量。

在大多数情况下, 书籍版面的设计者将给定要用到的各种粘连, 所以排版者不需要去确定粘连的各项是多大。例如, 对附录 B 的 plain TEX 的使用者而言, 当要在段之间输入额外的小空白时, 可以键入'`\smallskip`'; `\smallskip` 是 3 pt 的垂直粘连, 可以伸缩各 1 pt。下面就是一个 `\smallskip`:

当要想要非正常的间距, 又不想让各种粘连散布于整个文稿, 那么最好是用 point 明确给定它们, 通过象'`\smallskip`'这样的命令来表达你的意思。如果你需要更大或更小的间距, 那么 `\smallskip` 的定义很容易随之改变。Plain TEX 还给出了'`\medskip`'——它对应于两个 smallskip, 和'`\bigskip`'——它对应于两个 medskip。



在本手册中, plain TEX 的 `\medskip` 出现在每个"危险"标志段落的前后, 所以在你知晓之前已经看到了这种间距的很多例子。垂直粘连由'`\vskip`⟨glue⟩'生成, 其中 ⟨glue⟩ 是任意粘连类型。给出 ⟨glue⟩ 的一般方法是

⟨dimen⟩ `plus`⟨dimen⟩ `minus`⟨dimen⟩

其中, '`plus`⟨dimen⟩'和'`minus`⟨dimen⟩'是两个可选项, 如果没有就假定为零; '`plus`'后面跟可伸长量, '`minus`'后面跟可收缩量。例如, 附录 B 定义 `\medskip` 为'`\vskip6pt plus2pt minus2pt`'。粘连的自然间距量必须用 ⟨dimen⟩ 明确给出, 即使是零。



水平粘连也可以用同样方法得到, 但是要用 `\hskip` 来代替 `\vskip`。例如, plain TEX 定义 `\enskip` 为'`\hskip.5em\relax`'; 这个水平间距是一个"en", 即正好是当前字体的半个 em。在 `\enskip` 中没有伸缩量。'`.5em`'后的 `\relax` 的作用是, 当 `\enskip` 后面凑巧是'`plus`'或'`minus`'时, 防止 TEX 把它当成关键词。

在粘连伸缩不同比例时, 出现的一个有意思的事情是, 可能有*无限*伸长能力的粘连。例如, 再看看本章开头的四个盒子, 除了中间的粘连可伸长到无限远外, 其它都一样。现在, 总的伸长能力是无限大; 并且当行要变长时, 所有的额外空格都放在中间粘连中。例如, 如果要得到 58 的盒子, 中间粘连从 9 扩大到 15 个单位, 并且其它间距保持不变。

如果这样可无限伸长的粘连放在一排盒子的左边, 就是把它们"右对齐", 即把它们移动到所构造盒子的最右边界。如果你把两个可无限伸长的粘连团放在一个盒子的左右两边, 就是在把盒子列在一个大盒子中居中。这就是 \centerline 这个命令在 plain TeX 中的工作原理: 它把可无限伸长的粘连放在两端, 因此得到的是宽度为 \hsize 当前值的盒子。

第六章的 story 例子不但在居中时使用了无限大粘连, 而且在最后的 \vfill 指令中也用到了; 在本质上, '\vfill'的意思是"垂直跳过零, 但是可无限伸长"。换句话说, \vfill 用空白把当前页剩下的部分填满。

实际上, TeX 认识几种无限, 其中一些比其它的"更无限"。你可以用 \vfil 和 \vfill; 第二个比第一个更强。换句话说, 如果没有其它的无限出现, \vfill 将扩大以充满剩下的空间; 但是如果 \vfil 和 \vfill 同时出现, 那么 \vfill 就抑制了 \vfil 的伸长。你可以把 \vfil 看作一英里的伸长能力, 但是 \vfill 的能力是一万亿英里。

除了 \vfil 和 \vfill, TeX 还有 \hfil 和 \hfill, 它们是水平方向上的无限伸长。为了得到可无限收缩的粘连, 可以用 \hss 或 \vss。(名字'\hss'表示"水平伸缩", '\vss'表示"垂直伸缩"。) 最后, 原始控制系列 \hfilneg 和 \vfilneg 将抵消 \hfil 和 \vfil 的伸长能力; 后面我们将讨论这些古怪粘连的应用。

下面是一些 \hfil 的例子, 得到的是宽度为当前 \hsize 的行, 其中用到了 plain TeX 的宏 \line。

```
\line{This text will be flush left.\hfil}
\line{\hfil This text will be flush right.}
\line{\hfil This text will be centered.\hfil}
\line{Some text flush left\hfil and some flush right.}
\line{Alpha\hfil centered between Alpha and Omega\hfil Omega}
\line{Five\hfil words\hfil equally\hfil spaced\hfil out.}
```

▶ **练习12.2**
看看下面的结果是什么?

```
\line{\hfil\hfil What happens now?\hfil}
\line{\hfill\hfil and now?\hfil}
```

▶ **练习12.3**
下面三个宏有什么不同?

```
\def\centerlinea#1{\line{\hfil#1\hfil}}
\def\centerlineb#1{\line{\hfill#1\hfill}}
\def\centerlinec#1{\line{\hss#1\hss}}
```


为了得到这样的无限, 你可以在可伸长和可收缩的 ⟨dimen⟩ 中使用特殊单位'`fil`', '`fill`'和'`filll`'。例如, `\vfil`, `\vfill`, `\vss` 和 `\vfilneg` 其实分别等价于下面给出的粘连:

```
\vskip 0pt plus 1fil
\vskip 0pt plus 1fill
\vskip 0pt plus 1fil minus 1fil
\vskip 0pt plus -1fil
```

一般最好尽量坚持用第一阶无限(fil), 只有当要使用非常的无限时再使用第二阶(fill)。因此, 最高阶(filll) 总是紧急情况下作为最后一招使用。(因为不鼓励使用这个最高无限, 所以 TEX 没有给出原始控制系列'`\vfilll`'。) 你也可以使用无限的小数倍, 比如'`3,25fil`', 只要比 16384 fil 单位小就行。实际上, TEX 运算时采用的是 2^{-16} fil (或 fill 或 filll); 所以, `0.000007filll` 与 `0pt` 无法区分, 但是 `0.00001filll` 就比 `16383.99999fill` 更无限大了。 现在, 要告诉 TEX 的用户一些重要情况: Plain TeX 在每句的结尾加入了额外间距; 还有, 在标点后面它自动增加伸长能力(并且减小收缩能力)。理由是, 当要把一行充满到所要求的边界时, 在标点后面留更多的间距比在两个正常的单词中间留一般更好。例如, 看看下面从幼儿园找到的句子:

```
``Oh, oh!'' cried Baby Sally. Dick and Jane laughed.
```

如果 TEX 要让它保持自然宽度, 那么所有的间距是一样的, 但是引号和'Baby Sally.'后面除外:

"Oh, oh!" cried Baby Sally. Dick and Jane laughed.

但是如果行宽要扩大 5, 10, 15 points 或更多, TEX 得到的是:

"Oh, oh!" cried Baby Sally. Dick and Jane laughed.
"Oh, oh!" cried Baby Sally. Dick and Jane laughed.
"Oh, oh!" cried Baby Sally. Dick and Jane laughed.
"Oh, oh!" cried Baby Sally. Dick and Jane laughed.

逗号后面的粘连伸长的量是相邻单词之间粘连伸长量的 1.25 倍; 句点和 `!''` 后面的伸长量是其 3 倍。相邻的字母之间没有粘连, 所以各个单词看起来总是一样的。如果 TEX 必须把此行收缩到最小宽度, 结果是

"Oh, oh!" cried Baby Sally. Dick and Jane laughed.

在逗号后面的粘连的收缩是普通单词之间的 80%, 句点, 惊叹号或问号后面的粘连的收缩仅为其 1/3。

所有这些都是为了输出的好看, 但还是要为排版者增加一点负担, 因为 TEX 的确定句子结束的规则并不总是起作用。问题在于, 当象 ... 用作三个圆点的"省略号"时, 句点就在句子中间出现了。

还有, 如果要在一行上输入三个圆点来表示'. . .', 得到的就是'...'——圆点挨得太近了。解决之道就是转换到数学模式, 利用 plain TEX 中定义的 `\ldots` 控制系列。例如, 如果输入

```
Hmmm $\ldots$ I wonder why?
```

得到的就是'Hmmm . . . I wonder why?'。因为数学公式不遵守了正常的文本间距规则, 所以这看起来不错。第十八章详细讨论了 `\ldots` 及其相关内容。

缩写也会出现问题。例如, 在第六章的 story 说到'Mr. Drofnats'; 必须告诉 TEX, 'Mr.'、'Mrs.'、'Ms.'、'Prof.'、'Dr.'、'Rt. Hon.'等等后面的句点不能看作一个句子的结束。

通过输入'`Mr.~Drofnats`'我们就避免了第六章中出现的问题; "带子(tie)"标记 `~` 使得 plain TEX 插入一个正常空格, 并且避免在那个空格处断行。让 TEX 产生一个正常空格的另一种方法就是输入'`\␣`'(控制空格); 比如, '`Mr.\ Drofnats`'与'`Mr.~Drofnats`'几乎是一样的, 只是在'Mr.'后可能会断行。

带子标记适合于使用在名字缩写中, 还有几个其它常用的缩写, 象'Fig.'、'cf.'、'vs.和'resp.'; 你会发现自己很容易接受'`cf.~Fig.~5`'这样的训练。实际上, 在句中出现的常用缩写后面输入 `~` (而不是空格)是明智之举。排版手册会告诉你在缩写'e.g.'和'i.e.'后面总是要跟一个逗号, 而不是空格, 所以, 这些特殊情况不需要特别对待。

剩下的唯一经常出现的缩写是在参考文献中; 那里要使用控制空格。例如, 如果要得到'Proc. Amer. Math. Soc.', 要输入的是

```
Proc.\ Amer.\ Math.\ Soc.
```

当然要承认这个输入有点难看, 但是输出结果很好。当对付自以为聪明的计算机时, 这是我们经常使用的方法。

▶ **练习12.4**

看看怎样输入下列句子: "Mr. & Mrs. User were married by Rev. Drofnats, who preached on Matt. 19 : 3–9."

▶ **练习12.5**

把下列参考文献用 plain TEX 排出来: Put the following bibliographic reference into plain TEX language: Donald E. Knuth, "Mathematical typography," *Bull. Amer. Math. Soc.* **1** (1979), 337–372.

另一方面, 如果不需要这样的间距微调, 可以让 plain TEX 把所有的空格变成一样的, 而不管什么标点, 只需要在文稿的开头给出'`\frenchspacing`'。法语间距看起来是这样:

"Oh, oh!" cried Baby Sally. Dick and Jane laughed.

你还可以在这两种情况下变来变去, 规定了'`\nonfrenchspacing`'就得到复杂的间距, 规定了 `\frenchspacing` 就可以在某些局部编组内部得到同一间距。例如, 只有当输入某些文档的文献时, 可能要用到法语间距。

如果句点, 问号或惊叹号前面的字母是大写, 那么, TEX 不把这些标点看作句子的结束, 因为 TEX 假定这些大写字母大多是某些词的首字母。因此, 例如, 在'`Dr.~Livingstone~I.\ Presume`'中, '`I.`'后面的'`\`'是不必要的; 那个特殊的句点不会看作句子结束。

▶ **练习12.6**
怎样才能让 TEX 辨认出以大写字母结尾的句子(比如, '... launched by NASA.' or 'Did I?' or '... see Appendix A.')?

在第十一章, 我们主要讨论过的内部诊断格式中, 通过查看 hbox 的内容, 我们可以得到 TEX 在词之间所给出的粘连。 例如, 假定 `\nonfrenchspacing` 时, 在 TEX 消化及放在盒子中后, Baby Sally 的嚷嚷的开头如下:

```
.\tenrm \ (ligature ‘‘)
.\tenrm O
.\tenrm h
.\tenrm ,
.\glue 3.33333 plus 2.08331 minus 0.88889
.\tenrm o
.\tenrm h
.\tenrm !
.\tenrm " (ligature ’’)
.\glue 4.44444 plus 4.99997 minus 0.37036
.\tenrm c
.\tenrm r
.\tenrm i
.\tenrm e
.\tenrm d
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenrm B
.\tenrm a
.\tenrm b
.\kern-0.27779
.\tenrm y
```

```
.\glue 3.33333 plus 1.66666 minus 1.11111
.\tenrm S
.\tenrm a
.\tenrm l
.\tenrm l
.\tenrm y
.\kern-0.83334
.\tenrm .
.\glue 4.44444 plus 4.99997 minus 0.37036
```

在 `\tenrm` 字体中, 单词内部的正常粘连是 3.33333 pt, 再加上 1.66666 pt 的伸长和减去 1.11111 pt 的收缩。注意, 在本列表中, 标点后面的单词内部 `\glue` 伸长多, 收缩少; 并且在每个句子结束处的自然间距实际上更大。这个例子还提供了处理文本样本时 TEX 所做的几个其它事情: 它把 `‘‘` 和 `’’` 转变成单个字符, 即引号; 并且在这两个地方插入小的 kern 以优化间距。`\kern` 与粘连类似但不同, 因为 kern 不能伸缩; 还有, TEX 不会在 kern 处断行, 除非 kern 后面紧接着粘连。

你可能想知道 TEX 的单词内部粘连的实际规则到底是什么。例如, 当引号插入在下一个空格前时, TEX 是怎样得到各种参数的? 详细的过程有些巧妙, 但是不难理解。当 TEX 处理一个水平盒子和粘连列时, 首先有一个叫做当前“间距因子”的正整数。间距因子正常情况下是 1000, 表示单词内部粘连毋需改变。如果间距因子不是 1000,单词内部粘连按如下规则计算: 取当前字体的正常间距粘连, 并且当 $f \geq 2000$ 加上额外间距。(每个字体给出了一个正常间距, 正常伸长, 正常收缩和额外间距; 例如, 在 `cmr10` 中, 它们分别是 3.33333 pt, 1.66666 pt, 1.11111 pt 和 1.11111 pt。后面我们将更详细地讨论这些字体参数。) 接着, 伸长量就乘以 f/1000, 而收缩量乘以 $1000/f$。

但是, TEX 还有两个参数 `\spaceskip` 和 `\xspaceskip`, 它们允许你不使用当前字体的正常间距。如果 $f \geq 2000$ 并且 `\xspaceskip` 不是零, 那么单词内部间距使用 `\xspaceskip`。否则, 如果 `\spaceskip` 不是零, 使用 `\spaceskip` 粘连, 并且伸长和收缩量分别乘以 $f/1000$ 和 $1000/f$。例如, plain TEX 中, 宏 `\raggedright` 就是用 `\spaceskip` 和 `\xspaceskip` 去掉了所有单词内部的伸缩。

在水平列的开头, 间距因子为 1000, 并且在当前水平列上放置了非字符盒子或数学公式后, 它也设定为 1000。可以通过‘`\spacefactor=`⟨number⟩’为间距因子赋予任意特殊值; 但是一般地, 只有当列中出现一个简单的字符盒子时, f 的值才不是 1000。每个字符都有一个间距因子码, 当间距因子为 g 的字符出现在当前列时, 正常情况下, 就把 g 直接设置成新的间距因子。但是, 如果 g 是零, 那么 f 则不变; 如果 $f < 1000 < g$, 间距因子设置为 1000。(换句话说, f 不能一步从小于 1000 的值跳到大于 1000 的值。) 最大间距因子是 32767 (它比任何人想用的都大)。

当 `INITEX` 创建一种新的 TEX 类型时, 所有字符的间距因子码为 1000, 但是大写字母‘`A`’到‘`Z`’的代码为 999。(这个微小差别导致了大写字母后面的标点处在不同的情况下; 你知道为什么吗?) Plain TEX 用原始控制系列 `\sfcode` (它类似于 `\catcode`, 见附录 B)重新定义了几个这种代码; 例如, 下列指令

```
\sfcode‘)=0     \sfcode‘.=3000
```

使得右括号的间距因子“透明化”, 而句点后面的伸长量变成普通的三倍。`\frenchspacing` 的作用是把 `\sfcode`.` 重新设定为 1000。

当生成组合字时, 或者通过 `\char` 给出特殊字符时, 间距因子数要从组成组合字的单个字符计算出来。例如, plain TeX 设定右单引号的间距因子为零, 所以标点的效应会遗传下去。两个相邻的字符 `''` 组成了字符内部代码为 *'042* 的连字; 但是这个双右引号连字的间距因子数在 TeX 中没有设定, 所以 plain TeX 不给 `\sfcode'042` 赋予任何值。

▶ **练习12.7**

在例子 Dick-and-Jane 中, 每个记号后面的间距因子是什么?

当封装一个 hbox 时, TeX 如下来设置粘连: 盒子内容的自然宽度 x 等于内部盒子和 kern 的宽度以及内部所有粘连的宽度之和。还要算出盒子中的粘连伸缩总量; 比如, 总伸长量为 $y_0+y_1\,\mathrm{fil}+y_2\,\mathrm{fill}+y_3\,\mathrm{filll}$, 总收缩量为 $z_0+z_1\,\mathrm{fil}+z_2\,\mathrm{fill}+z_3\,\mathrm{filll}$。现在, 把自然宽度 x 与要求的宽度 w 相比较。如果 $x=w$, 那么所有的粘连设置为其自然宽度。否则就要改变粘连, 按下面的方法计算出“粘连调整比例”和“粘连调整阶次”: (a). 如果 $x<w$, TeX 就要把盒子的内容变长; 粘连阶次是最大的非零 y_i 的下标 i, 粘连比例是 $r=(w-x)/y_i$。(如果 $y_0=y_1=y_2=y_3=0$, 那么就没有伸长量; i 和 r 都是零。) (b). 如果 $x>w$, TeX 就要类似把盒子的内容缩短; 粘连阶次是 $z_i\neq 0$ 的最大下标 i, 正常的粘连比例是 $r=(x-w)/z_i$。但是, 如果 $i=0$ 且 $x-w>z_0$, 那么 r 设定为 1.0, 这是因为不能超出最大收缩量。(c). 最后, 修改要包装起来的水平列的每个粘连团。假定粘连是自然宽度 u, 伸长量为 y, 收缩量为 z, 其中 y 是第 j 阶无限大, z 是第 k 阶无限大。那么, 如果 $x<w$ (要伸长), 那么如果 $j=i$, 这个粘连的新宽度为 $u+ry$; 如果 $j\neq i$, 它保持自然宽度。如果 $x>w$ (要收缩), 那么如果 $k=i$, 这个粘连的新宽度为 $u-rz$; 如果 $k\neq i$, 它保持自然宽度。注意, 只有当粘连的最高阶无限大不抵消时, 才出现伸缩。

如果你给出命令‘`\hbox to` ⟨dimen⟩`{`⟨contents of box⟩`}`’, 那么 TeX 将构造一个所给定宽度 w 的 hbox, 其中 w 是 ⟨dimen⟩ 的值。例如, 在本张前面讨论的宏 `\line` 的定义就是‘`\hbox to\hsize`’。TeX 还允许你给出精确的伸长量或收缩量; 命令‘`\hbox spread`⟨dimen⟩`{`⟨contents of box⟩`}`’得到的盒子的宽度 w 就是盒子内容的自然宽度加上所给定的量。例如, 本章前面的展示的一个盒子就是如下方法得到的:

```
\hbox spread 5pt{``Oh, oh!'' ... laughed.}
```

在简单的情形下, 当只要得到自然宽度的盒子时, 不必使用‘`\hbox spread 0pt`’; 可以直接用‘`\hbox{`⟨contents of box⟩`}`’。

所构建盒子的基线是内部小盒子的公共基线。(更确切地说, 如果小盒子没有被升降, 那么就是它们共有的公共基线。) 所构建盒子的高度和深度分别由基线上下的内部小盒子的最大距离确定。所得的 `\hbox` 的高度和深度不能是负值, 但是宽度可以是负的。

▶ **练习12.8**

假定 `\box1` 的高度为 1 pt, 深度为 1 pt, 宽度为 1 pt; `\box2` 的高度为 2 pt, 深度为 2 pt, 宽度为 2 pt。第三个盒子如下得到:

```
\setbox3=\hbox to3pt{\hfil\lower3pt\box1\hskip-3pt plus3fil\box2}
```

那么, `\box3` 的高度, 深度和宽度各是什么? 找出 box1 和 box2 相对于 box3 的基准点的位置。

设置 vbox 的粘连的过程类似于 hbox 的; 但是在讨论 `\vbox` 这个命令之前, 需要讨论一下 TEX 怎样把盒子垂直堆砌起来, 才能使它们的基线的间隔为固定的距离。水平列中的盒子一般互相紧接着, 但是对垂直列却不行; 试想一下, 如果不管所输入的行是否包含高的字母, 或者包含比基线更低的字母, 而把它们紧密地放在一起, 会有多难看。

TEX 对这个问题的解决方法包括了三个原始控制系列, 它们叫做 `\baselineskip`, `\lineskip` 和 `\lineskiplimit`。格式的设计者如下选择这三个量的值:

`\baselineskip=`⟨glue⟩
`\lineskip=`⟨glue⟩
`\lineskiplimit=`⟨dimen⟩

其原理为: 只要盒子被加入垂直列中, TEX 将插入“行间粘连”, 使得新盒子的基线与前一个盒子的基线之间的距离正好等于 `\baselineskip` 的值。但是, 如果这样得到的行间粘连导致新盒子的上边界与前一个盒子的下边界的距离小于 `\lineskiplimit`, 那么, 行间粘连就变成了 `\lineskip`。换句话说, 只要盒子挨得不太近, 相邻基线之间的距离设置为 `\baselineskip`; 若太近了, 就用 `\lineskip` 把相邻的盒子分开。

前一段的行间粘连法则在执行时不考虑可能出现的其它类型的粘连; 由 `\vskip` 和 `\kern` 明确给出的所有垂直间距都与行间粘连无关。因此, 例如, 两行之间的 `\smallskip` 设定其基线之间距离比普通情况大 `\smallskip` 那样大的量; 不管这些行之间是否使用 `\lineskip`, 它都不受影响。

例如, 假定 `\baselineskip=12pt plus 2pt`, `\lineskip=3pt minus 1pt` 和 `\lineskiplimit=2pt`。(这些值不是特别有用; 只是选用它们来说明一下规则。) 再假定深度为 3 pt 的盒子是最后加入到堆砌垂直列中的; 现在要加入一个高度为 h 的盒子。如果 $h = 5$ pt, 行间粘连就是 4 pt plus 2 pt, 这是因为把 h 和前面的深度与行间粘连相加时, 我们得到基线间隔为 12 pt plus 2 pt。但是如果 $h = 8$ pt, 行间粘连为 3 pt minus 1 pt, 因为当忽略伸缩时, 为了不破坏给定的 `\lineskiplimit`, 所以选择的是 `\lineskip`。

当要排版一个多页的文档时, 一般最后把 `\baselineskip` 定义为不可伸缩, 因为这样得到的页面更一致。基线之间小的间隔变化——比如只有半 point——也可导致排版的页面在差别很大, 因为它强烈地影响黑白比例。另一方面, 如果你要排版单页文档, 可能要用到可伸长的 baselineskip, 使得 TEX 帮着排满页面。

▶ **练习12.9**
怎样设置 `\baselineskip`, `\lineskip` 和 `\lineskiplimit` 才能使得行间粘连是下面的盒子高度的“连续”函数(即, 当盒子高度只有微小变化时, 行间粘连不会产生跃变)?

讨论一下 TEX 内部的盒子和粘连的表示对巩固这些原理是有益的。下面是排版本段(译注: 英文原版)时 TEX 构造的垂直列信息摘录:

```
\glue 6.0 plus 2.0 minus 2.0
\glue(\parskip) 0.0 plus 1.0
\glue(\baselineskip) 1.25
\hbox(7.5+1.93748)x312.0, glue set 0.80154, shifted 36.0 []
\penalty 10000
\glue(\baselineskip) 2.81252
\hbox(6.25+1.93748)x312.0, glue set 0.5816, shifted 36.0 []
\penalty 50
\glue(\baselineskip) 2.81252
\hbox(6.25+1.75)x348.0, glue set 116.70227fil []
\penalty 10000
\glue(\abovedisplayskip) 6.0 plus 3.0 minus 1.0
\glue(\lineskip) 1.0
\hbox(149.25+0.74998)x348.0 []
```

本例的第一个 `\glue` 是每个"危险"标记段前的 `\medskip`。接着是 `\parskip` 粘连, 它自动加在新段的第一行前面。接下来是一些 1.25 pt 的行间粘连; 它等于总数 11 pt 减去下一个盒子的高度 (7.5 pt) 和前一个盒子的深度。(前一个盒子没有显示出来——它是 exercise 12.9 的底线——但是我们可以推出来它的深度是 2.25 pt。) 紧接的 `\hbox` 是本段的第一行; 由于悬挂缩进, 它向右移动了 36 pt。这个 hbox 的粘连调整比例是 0.80154; 即, 内部的粘连伸长为其伸长能力的 80.154%。(在收缩时, '`glue set`'比例的前面有一个'`-`'; 所以我们知道了, 这里包括了伸长。) 如果盒子中的某些东西没有显示, TEX 将在每个 hbox 行后面放上'`[]`'。(如果 `showboxdepth` 设定的值更高, 那么盒子的内容将全部显示出来。) `\penalty` 命令用来防止不适当的换页, 我们在后面将讨论它。第三个 hbox 的粘连比例是 116.70227, 其中单位是一阶无限大伸长(即, fil); 这来自于在所显示内容前面暗中插入的 `\hfil`, 以充满本段的第三行。最后, 对于高度为 149.25 pt 的大盒子, 用 `\lineskip` 来设置行间粘连。这个大盒子包含了 typewriter 字体显示的各个行; 它们被包装在一个大盒子中, 在页面中不能被拆开。仔细分析本例, 就能得到 TEX 内部运作的很多信息。

例外: 在标尺盒子前后不插入行间粘连。在盒子之间使用 `\nointerlineskip` 就可以去掉行间粘连。

TEX 的行间粘连的命令包括另一个原始控制系列 `\prevdepth`, 它一般包含的是堆砌垂直列上最后一个盒子的深度。但是, `\prevdepth` 在垂直列开头或标尺盒子后面设定为标志值 -1000 pt; 这是为了去掉下一个行间粘连。在建立垂直列时, 用户可以在任何时候改变 `\prevdepth` 的值; 因此, 例如, 附录 B 的宏 `\nointerlineskip` 就是直接定义为'`\prevdepth=-1000pt`'。

下面是 TEX 计算盒子间行间粘连的具体方法: 假设高度为 h 的新盒子(不是标尺盒子)要追加在当前垂直列的底部, 并且设 `\prevdepth` $= p$, `\lineskiplimit` $= l$, `\baselineskip` $= (b \text{ plus } y \text{ minus } z)$。如果 $p \le -1000$ pt, 那么不插入行间粘连。否则, 如果 $b-p-h \ge 1$, 行间粘连'$(b-p-h)$

plus y minus z'将插入在新盒子上方。或者插入 \lineskip 粘连。最后，\prevdepth 设置为新盒子的深度。

▶ **练习12.10**
用户 Mr. B. L. 要把许多盒子放在一个垂直列中，并且它们之间不能有间距。他不想在每个盒子后面添加 \nointerlineskip，所以决定把 \baselineskip, \lineskip 和 \lineskiplimit 都设置为 0pt。这样可以吗？

在垂直情形下类似于 \hbox 的是 \vbox，类似于水平情形，TeX 按类似方法提供命令'\vbox to⟨dimen⟩'和'\vbox spread⟨dimen⟩'。但是，因为在水平方向上只有宽度，而在垂直方向上有深度和高度，所以要复杂一些。在 \vbox 的命令中的尺寸指的是 vbox 最后的高度，所以，例如，'\vbox to 50pt{...}'得到的盒子是 50 pt 高；这是因为 vbox 内部可伸缩的各种东西都出现在高度部分，而深度不受粘连调整的影响。

所构造的 \vbox 的深度最后看作内部底盒子的深度。因此，在概念上，vbox 是这样构建的：取一组盒子，把它们的基准点垂直对齐放好；接下来，把最低的盒子的基准点取为整个盒子的基准点，调整粘连使得最后的高度为给定高度。

但是，vbox 的这个陈述掩盖了特殊情形下出现的一些特性。例如，通过'\moveright⟨dimen⟩⟨box⟩'或者'\moveleft⟨dimen⟩⟨box⟩'，你可以把垂直列中的盒子向左向右移动；这类似于水平列中的 \raise 或 \lower 盒子的功能，它还暗示 vbox 中的基准点不总是在一条垂线上。还有，必须防止盒子变得太深，以免它们超出页面边界太多；在后面的章将看到，除了盒子和粘连，垂直列还有象 penalty 和 mark 等其它属性。

因此，所构造盒子的深度的实际规则稍微 TeX 化了一点。如下：给出要通过 \vbox 包装起来的垂直列，要确定的是其自然深度。(1)．如果垂直列没有包含盒子，深度为零。(2) 如果至少有一个盒子，但是如果最后一个盒子后面跟着 kern 或粘连，可能还插入了 penalty 或其它内容，那么深度为零。(3)．如果至少有一个盒子，且最后的盒子没有跟 kern 或粘连，那么深度为此盒子的深度。(4)．但是，如果通过规则 (1), (2), (3) 计算得到的深度超过了 \boxmaxdepth，那么深度为当前 \boxmaxdepth 的值。(Plain TeX 把 \boxmaxdepth 设置为可能的最大尺寸；因此，规则 (4) 一般用不到，除非你把它变成更小的值。当的确要用规则 (4) 来减小深度时，TeX 把超出的深度变成盒子的自然高度，本质上就是把基准点向下移动，直到深度减小到给定的深度。)

在 vbox 中计算粘连与在 hbox 中一样，通过自然高度 x 和要求的高度 w 的差和给出的伸缩能力来确定粘连调整比例和粘连调整阶次。

所计算的 \vbox 的宽度是所包装的盒子延伸到基准点右边的最大距离，要算上可能的偏移。这个宽度总是非负的。

▶ **练习12.11**
假设 \box1 的高度为 1 pt，深度为 1 pt，宽度为 1 pt；\box2 的高度为 2 pt，深度为 2 pt，宽度为 2 pt；baselineskip, lineskip, 和 lineskiplimit 都是零；并且 \boxmaxdepth 很大。第三个盒子定义为

```
\setbox3=\vbox to3pt{\moveright3pt\box1\vskip-3pt plus3fil\box2}
```

那么, `\box3` 的高度, 深度和宽度各是什么? 给出相对于 box3 的基准点的 box1 和 box2 的基准点。

▶ 练习12.12
在前一个练习的假设下, 修改 `\baselineskip=9pt minus3fil`, 并定义 `\box4` 为

```
\setbox4=\vbox to4pt{\vss\box1\moveleft4pt\box2\vss}
```

回答上面的同样问题。

▶ 练习12.13
修改 `\boxmaxdepth=-4pt` 后回答上面的问题。

我们看到, `\vbox` 把一组盒子组合成更大的盒子, 其基线与内部底盒子的基线一致。TEX 提供了另外一个叫做 `\vtop` 的命令, 它给出象 `\vbox` 一样的盒子, 但是基线与内部顶盒子的基线一致。例如,

```
\hbox{Here are \vtop{\hbox{two lines}\hbox{of text.}}}
```

得到

Here are two lines
 of text.

可以象在 `\vbox` 中那样使用‘`\vtop to`⟨dimen⟩’和‘`\vtop spread`⟨dimen⟩’, 但是你应该知道这样的指令的意思。TEX 如下执行 `\vtop`: (1). 首先, 把 `\vtop` 看成和 `vbox` 一样生成一个垂直盒子, 规则与 `\vbox` 相同。(2). 如果新盒子内部第一个项不是盒子, 那么最后的高度 x 为零; 如果是, x 就是此盒子的高度。(3). 把 h 和 d 设定为第一步中 vbox 的高度和深度。如果需要, TEX 将上下移动基准点, 使得盒子的高度为 x, 深度为 $h + d - x$。

▶ 练习12.14
看看从‘`\vbox to`⟨dimen⟩`{}`’和‘`\vtop to`⟨dimen⟩`{}`’得到的空盒子。其高度, 深度和宽度各是什么?

▶ 练习12.15
定义一个宏 `\nullbox#1#2#3`, 它生成高度, 深度和宽度分别为这三个参数的盒子。 在输出时, 盒子什么也不显示出来。

`\vbox` 倾向于生成大高度小深度的盒子, 而 `\vtop` 倾向于生成小高度大深度的盒子。但是, 如果要从大盒子制作一个垂直列, 可能对 `\vbox` 和 `\vtop` 都不满意; 可能想要有两个基准点的盒子, 一个在顶部, 一个在底部。如果使用双基准点的方法, 就可以在垂直列中, 把上一个盒子的下基准点和下一个盒子的上基准点定义为行间粘连。但是, 唉, TEX 只对每个盒子给出了一个基准点。

有一个办法来达到此目的, 利用称为“strut(支架)”的一个重要原理。Plain TEX 把 `\strut` 定义为宽度为零的不可见盒子, 它正好延伸到基线上下, 这样, 如果每行都包含一个 strut, 那么就根本不需要行间粘连。在 plain TEX 中, 基线间隔为 12 pt; 其 `\strut` 为高度为 8.5 pt, 深度为 3.5 pt, 宽度为 0 pt 的垂直标尺。) 如果要在大 vbox 中的顶行上放一个 strut, 并且在底行上放另一个 strut, 那么, 通过直接把这些大盒

子叠在一起就可以在一个更大的集合中得到适当的间距。例如, 在附录 B 的宏 `\footnote` 把 strut 放在每个 footnote 的开头和结尾, 在某些页面底部有几个 footnote 时, 就得到正确的间距了。

如果你理解了盒子和粘连, 就可以看懂 plain TEX 的宏 `\rlap` 和 `\llap` 了; 这些名字是"右重叠"和"左重叠"的缩写。给出'`\rlap{`⟨something⟩`}`'就象排版 ⟨something⟩, 接着有退回来, 好像还没有排版什么东西一样。更确切地说, '`\rlap{`⟨something⟩`}`'生成一个宽度为零的盒子, 而'⟨something⟩'正好出现在盒子的右边(但是不占任何空间)。`\llap` 类似, 只是先退格; 换句话说, '`\llap{`⟨something⟩`}`'生成一个宽度为零的盒子, 而'⟨something⟩'就在盒子的左边。例如, 利用 typewriter 字体, 可以通过'`\rlap/=`'或'`/\llap=`'得到'≠'。可以利用 `\llap` 把文字放在左边界中, 或者利用 `\rlap` 放在右边界中, 因为 TEX 并不把盒子的内容总限制在盒子的边界内。

有关 `\rlap` 和 `\llap` 有意思的是, 它们可以如此简单地通过无限粘连得到。`\rlap` 的一个定义是

```
\def\rlap#1{{\setbox0=\hbox{#1}\copy0\kern-\wd0}}
```

但是不需要很多长度运算。附录 B 中的实际定义更高超, 即,

```
\def\rlap#1{\hbox to 0pt{#1\hss}}
```

并且它值得好好想一想。例如, 假设要使用 `\rlap{g}`, 其中字母 g 的宽度为 5 pt。因为 `\rlap` 制作的盒子的宽度为 0 pt, 由 `\hss` 表示的粘连必须收缩 5 pt, 好了, 粘连把 0 pt 当成其自然宽度, 但是它有无限收缩能力, 所以可以很容易收缩 −5 pt; 并且'`\hskip-5pt`'正好是这种情况下 `\rlap` 想要的。

▶ **练习12.16**
不偷看附录 A 或 B, 猜一下 `\llap` 的定义。

▶ **练习12.17**
(这是 exercise 12.2 的答案, 但是是一个窍门。) 看看下面的结果:

```
\line{\hfil A puzzle.\hfilneg}
```

13. 模式

正如人们有不同心情一样, TEX 也有不同的“模式”。(只是 TEX 比人更容易掌握。) 其中有六种模式:

- 垂直模式。[构建主垂直列, 输出的页面源于此。]
- 内部垂直模式。[构建 vbox 的垂直列。]
- 水平模式。[构建段落的水平列。]
- 受限水平模式。[构建 hbox 的水平列。]
- 数学模式。[构建数学公式, 并放在水平列中。]
- 列表数学模式。[构建数学公式, 并暂时中断当前段, 单独放在一行上。]

在简单情况下, 你不需要知道 TEX 处在什么模式中, 因为计算机正在正常工作。但是, 如果你看见错误信息说‘`! You can’t do such-and-such in restricted horizontal mode`’, 那么模式方面的知识有助于弄明白 TEX 认为出了什么问题。

一般说来, 当 TEX 正在处理的是页面上一列互相叠放的盒子和粘连时, 就处在一种垂直模式中; 当处理的是沿基线对齐的一列紧挨着的盒子和粘连时, 就处在一种水平模式; 当遇到公式时, 就处在一种数学模式。

一个有代表性的 TEX 工作的信息说明就可以把模式的原理搞清楚: 在开头, TEX 处于垂直模式, 准备好构造页面。如果当 TEX 处在垂直模式时你给出粘连或盒子, 那么粘连或盒子就放在已经排好的当前页面的下面。例如, 在我们讨论过的第六章的样例中, `\vskip` 指令把垂直粘连放在页面中; `\hrule` 在 story 的顶部和底部放上水平标尺。`\centerline` 命令得到的也是包含在垂直列中的盒子; 但是那些盒子所需要的处理比标尺盒子要多一点: 当遇到‘`\centerline{\bf A SHORT STORY}`’时, TEX 处在垂直模式, 当处理词语‘`A SHORT STORY`’时, 它临时转到受限水平模式; 然后, 在设置好 `\centerline` 的粘连后, 处理程序回到垂直模式。

继续讨论第六章的例子, 一旦遇到‘`Once upon a time`’的‘`O`’, TEX 就转到水平模式。水平模式是构建段落的模式。这个段落(story 文件的第 7 到第 11 行)是在水平模式下输入的; 接着文本被分割为相应宽度的输出行, 这些行被放在盒子中并添加到页面(它们之间要加上相应的行间粘连), 最后, TEX 回到垂直模式。第 12 行的‘`M`’再次启动水平模式。

当 TEX 处在垂直模式或内部垂直模式时, 新段落的第一个记号在段落期间改变为水平模式。换句话说, 非垂直定向的东西把模式自动从垂直变成水平了。当你输入任何字符, `\char`, `\accent`, `\hskip`, `\␣`, `\vrule` 或数学转换符 (`$`) 时, 就发生这种情况; TEX 将插入当前段落的缩进, 并且好像它已经处在水平模式一样重新读入水平记号。

也可以明确告诉 TEX 要进入水平模式, 而不是靠这些暗中进行的模式转换, 只要使用'\indent'或'\noindent'即可。例如, 如果第六章的 stroy 文件的第 7 行的开头为

```
\indent Once upon a time, ...
```

那么得到的是同样的输出, 因为'\indent'将告诉 TEX 要开始一段。还有, 如果行的开头为

```
\noindent Once upon a time, ...
```

那么 story 的第一段就不缩进。如果当前模式为垂直或内部垂直模式, 那么命令 \noindent 直接告诉 TEX 要进入水平模式; \indent 也类似, 但是它还产生一个宽度为 \parindent 的当前值的空盒子, 并把它放在当前水平列中。Plain TEX 设置 \parindent=20pt。如果你输入 \indent\indent, 就得到双倍缩进; 如果输入 \noindent\noindent, 那么第二个 \noindent 就不起作用。

▶ **练习13.1**
如果在水平模式输入'\hbox{...}', TEX 就构建给出的盒子, 并把它放在当前段。类似地, 如果在垂直模式输入'\hbox{...}', TEX 就构建盒子, 并把它放在当前页面。如果要用 \hbox 开始一个新段, 应该怎样做?

当处理简单文稿时, TEX 几乎把所有时间都花在水平模式(构建段落)中, 在垂直模式(段落之间)只有少量时间。当输入 \par 或者空行时, 一个段落就结束了, 因为安装第八章的读入规则, 空行将被转换成 \par。当输入与水平模式不相容的某些东西时, 段落也要结束。例如, 在第六章的 story 文件的第 16 行, 命令'\vskip 1in'也足以结束与'...beautiful documents.'有关的这段了; 不需要 \par, 因为 \vskip 要插入不能在段落中出现的垂直粘连。

如果在水平模式中出现数学模式开始的记号($), 那么 TEX 将进入数学模式并处理到结束符'$'的公式, 接着把此公式的文本添加到当前段落, 返回水平模式。因此, 在第十二章的"I wonder why?"例子中, TEX 在处理 \ldots 时临时进入数学模式, 把 dot 看作公式。

但是, 如果在段落中出现两个紧接的数学模式开始的记号($$), TEX 将在此中断段落, 把至此的段落放在要封装的垂直列中, 接着在列表数学模式下处理数学公式, 再把这个公式也放在要封装的垂直列中, 最后, 回到水平模式, 处理段落后面的内容。(要列表的公式要以'$$'结束。) 例如, 如果输入

```
the number $$\pi \approx 3.1415926536$$ is important.
```

TEX 就在 $$ 之间进入列表数学模式, 并且所得到的输出如下: the number

$$\pi \approx 3.1415926536$$

is important.

当处在垂直或内部垂直模式时, TEX 将忽略掉空格和空行(或者 \par), 所以不必担心这些东西会改变模式或影响排版结果。但是, 控制空格(\␣)将被看作一个段落的开头; 在缩进后, 此段落

将以空格来开始。

在 TeX 文稿的结尾, 一般最好输入'\bye'以结束所有内容, 其意思是'\vfill\eject\end'。其中, 'vfill'把 TeX 转换到垂直模式, 并插入足够的空白来充满最后一页; '\eject'把最后一页输出; '\end'告诉计算机进入终结程序。

当要从盒子的垂直列(使用 \vbox, \vtop, \vcenter, \valign, \vadjust 或 \insert时)构建输出时, TeX 进入内部垂直模式。当要从盒子的水平列(使用 \hbox 或 \halign 时)构建输出时, 进入受限水平模式。盒子的构建在第十二和二十一章中讨论。后面我们将看到, 在内部垂直模式和普通垂直模式之间, 在受限水平模式和普通水平模式之间只有很小的差别; 但是它们却不雷同, 因为目的不同。

只要 TeX 遇见一个输入记号, 要确定下一步应该怎样做时, 当前模式都潜在地影响此记号的意义。例如, 在垂直模式中, \kern 生成一个垂直间距, 但是在水平模式生成一个水平间距; 象'$'这样的数学转换符在水平模式中要进入数学模式, 但是在数学模式时, 其作用是退出数学模式; 两个紧接的数学转换符($$)在水平模式中要进入列表数学模式, 但是在受限水平模式中它们只表示一个空白数学公式。某些命令在某些模式下不相称, 利用这个情况, TeX 就可以帮助你把潜伏在文稿中的错误进行修复。第二十四和二十六章详细讨论了每个可能的记号在每个模式中的作用。

TeX 常常在一个模式中中断其工作以在另一个模式执行其它任务, 在执行完毕后, 将重新恢复到原来的模式。例如, 可以在任何模式中输入'\hbox{'; 当 TeX 要处理它时, 就把正在处理的东西搁置起来, 并且进入限制水平模式。最后匹配的'}'将结束 hbox, 于是搁置起来的工作又重新进行。在这种情况下, TeX 可以同时处在很多模式中, 但是只有最内部的模式在当时起作用; 其它模式被 TeX 暂时忘记了。

要理解 TeX 的模式, 就看看下面这个古怪的测试文件 modes.tex, 它同时包含了所有的模式:

```
\tracingcommands=1
\hbox{
$
\vbox{
\noindent$$
x\showlists
$$}$}\bye
```

modes.tex 的第一行告诉 TEX 记录下接到的每个命令; 只要 \tracingcommands 是正的, TEX 就给出诊断信息。的确, 如果编译 \modes.tex, 就得到了文件 modes.log, 它包含下列信息:

```
{vertical mode: \hbox}
{restricted horizontal mode: blank space}
{math shift character $}
{math mode: blank space}
{\vbox}
{internal vertical mode: blank space}
{\noindent}
{horizontal mode: math shift character $}
{display math mode: blank space}
{the letter x}
```

意思是, TEX 首先在垂直模式下遇见一个记号 \hbox; 这使其继续进行并且在幕后读入'{'。接着 TEX 进入受限水平模式, 并且遇见来自文件的第二行结尾的空格记号。接着遇见数学转换符记号(仍然处在受限水平模式), 它使得转换到数学模式; 接下来是另一个空格。再接着 \vbox 导致进入内部垂直模式, 并且 \noindent 在内部开始了水平模式; 两个紧接的 $ 号导致进入列表数学模式。 (只有第一个 $ 被 \tracingcommands 显示出来, 这是因为它使得 TEX 到前面去找另一个。)

在上面的输出后面, modes.log 的内容是'{\showlists}'。这是另一个处理诊断的命令, 可以用来显示出 TEX 一般不显示的信息; 它使 TEX 把正在运行的东西列表显示——在当前模式下和在被搁置起来的所有包裹在外面模式下:

```
### display math mode entered at line 5
\mathord
.\fam1 x
### internal vertical mode entered at line 4
prevdepth ignored
### math mode entered at line 3
### restricted horizontal mode entered at line 2
\glue 3.33333 plus 1.66666 minus 1.11111
spacefactor 1000
### vertical mode entered at line 0
prevdepth ignored
```

在本情况下, 列表给出了五个等级的活性, 它们都出现在 modes.tex 第 6 行的结尾。当前模式首先列出来, 即, 列表数学模式, 它在第 5 行开始。当前数学列表包含一个"mathord"对象, 由第 1 族中的字母 x 组成。(耐心一点, 当学到 TEX 的数学公式那章时就理解它的意思了。) 列表数学模式外面是内部垂直模式, 当包含列表公式的段落结束后, TEX 将回到它。在此等级上, 垂直列表是空的; '\prevdepth ignored'意思是 \prevdepth 的值 ≤ -1000 pt, 因此下一个行间粘连将被去掉(参见第十二章)。这个内部垂直模式外面的数

学模式同样也是空列, 但是数学模式外面的受限水平模式包含一些粘连。最后是主要垂直模式, 它包裹了所有的东西; 这个模式‘`entered at line 0`’, 即在文件 `modes.tex` 输入之前; 在这个最外面的等级上的垂直列迄今为止什么也没有。

▶ **练习13.2**

为什么在这些列表中的一个中有粘连, 而其它的没有?

▶ **练习13.3**

在 `\showlists` 的输出后, `modes.log` 文件包含来自 `\tracingcommands` 的更多输出。实际上, 此文件的下两行是

```
{math shift character $}
{horizontal mode: end-group character }}
```

因为在第 7 行的‘`$$`’结束了列表公式, 并且回到了中断了的水平模式。你知道 `modes.log` 文件的下三行是什么吗?

▶ **练习13.4**

假定 TEX 生成了一个文档, 却没有离开过垂直模式。这样的文稿是怎样的呢?

▶ **练习13.5**

有些 TEX 模式不能直接包在其它模式中; 例如, 列表数学模式不能直接包含在水平模式中, 即使列表出现在段落中, 这是因为在开始处理列表公式时, 水平模式的搁置起来的当前段落总是要结束掉, 并且从 TEX 的内存中清除掉。看看在 `\showlists` 输出中出现的两个紧接着的模式之间, 所有的关系有什么特征?

14. 分段为行

排版系统的一个主要任务就是把一长列单词分割成适当长度的各个行。例如, 本手册的每段被分为 29 个字宽的行, 但是当输入文稿时, 作者却不用去理会这些细节。 TEX 把每段看作一个整体, 用一种有趣的方法来选择出断点; 实际上, 段落的最后一个单词也能对第一行的排版产生影响。结果是, 单词之间的间距尽可能一致, 并且计算机要尽量避免把单词或公式放在不同行。

第六章的实战遇见说明了一般的思想: 我们讨论了"badness"这个概念, 并且在不同情况下得到了"溢出"和"松散"的盒子。我们还看到, 设置不同的 TEX 的 `\tolerance` 的参数将得到不同的结果; 容许度越高, 容许的间距越大。 遵循最小化 badness 的思想, TEX 要找到排版任意段落的最好方法。但是, 这样的"badness"没有考虑到所有问题, 而且如果完全依靠自动排版, 常常会见到在感觉上并不真是最好的断行; 这是不可避免的, 因为计算机不能理解人类做事的方式(至少现在还没有)。因此, 有时候要告诉计算机, 有些地方不能插入断点。反过来, 有时要在一个特殊点强制插入断点。 TEX 给出了一个便捷的方法来避免感觉上不好的断行, 因此, 通过直接给计算机几个提示, 就可以得到最好的结果。

"带子"——在 plain TEX 用'~'表示——是得到圆满断行的关键。一旦学会了怎样插入它, 你就从一般排版等级晋升到了中级排版专家了。而且, 当输入文稿时, 几乎不用思考, 就可以轻松插入一般的带子。

[译者注]: 在处理中文时, 这样定义的带子是没有什么用处的。因此, CJK 和 CCT 中都已经把 ~ 重新定义为一个可伸缩的粘连, 用在中文和英文之间。具体定义如下:

```
\global\def~{\hskip 0.25em plus 0.125em minus 0.08em \ignorespaces}
```

当输入 ~ 时, 它与输入空格一样, 只是 TEX 不在此空格断行而已。还有, 不要在 ~ 后面留下空白, 因为会把它们看作额外的空格。如果在输入文件的每行的结尾都放上 ~, 就会得到比更预想更宽的间距, 因为 ~ 后面的 ⟨return⟩ 得到了一个额外空格。

在第十二章我们已经看到, 不在句子结尾出现的略语后面加上 ~ 一般都不错。在下面几个其它地方也要加上带子:

- 文档中涉及到命名的部分:

```
Chapter~12              Theorem~1.2
Appendix~A              Table~\hbox{B-8}
Figure~3                Lemmas 5 and~6
```

(在最后的例子中, '`Lemmas`'后面没有添加 ~, 因为'5 and 6'出现在行首是可以接受的。`\hbox` 的使用在下面讨论。)

- 在人名之间和多个姓之间:

```
Donald~E. Knuth                       Luis~I. Trabb~Pardo
Bartel~Leendert van~der~Waerden       Charles~XII
```

注意, 有时候连字化名字比在单词之间断开要好; 比如, 'Don-'和'ald E. Knuth'比'Donald'和'E. Knuth'的回旋余地要大。前一个规则可以看作此规则的特殊情形, 因为可以把'Chapter 12'看作复合名字; 另一个例子是'`register~X`'。有时候, 名字太长, 使得我们不敢把它们绑在一起, 免得无法断行。

```
Charles Louis Xavier~Joseph de~la Vall\'ee~Poussin.
```

- 在数学符合与其名称之间:

```
dimension~$d$       width~$w$       function~$f(x)$
string~$s$ of length~$l$
```

但是, 最后一个例子可以与下列情况比较一下:

```
string~$s$ of length $l$~or more.
```

- 在一列符号之间:

```
1,~2, or~3
$a$,~$b$, and~$c$.
1,~2, \dots,~$n$.
```

- 当符号与前面的内容密切相关时:

```
of~$x$
from 0 to~1
increase $z$ by~1
in common with~$m$.
```

但是此规则不要应用在复合对象上:

```
of $u$~and~$v$.
```

- 当数学惯用语出现在句子中时:

```
equals~$n$    less than~$\epsilon$    (given~$X$)
mod~2         modulo~$p^e$            for all large~$n$
```

比较一下'`is~15`'和'`is 15~times the height`'。

- 当在段落中列举各种情况时:

```
(b)~Show that $f(x)$ is (1)~continuous; (2)~bounded.
```

把所有这些规则提炼为一到两个简单的原理就好了, 而且如果可以自动遵守规则而不需要键盘输入就更好了; 但是其中出现了很多敏感的语义问题。因此, 在带子上最好你自己去取舍。计算机需要你的帮助。

带子可以防止 TEX 在空格处断行, 但是有时候要阻止计算机造字连字或破折号处断行。这时要使用 `\hbox`, 因为 TEX 不会分割盒子的内容; 盒子一旦构建出来, 就是不可分割的单位。我们已经在早先讨论的'`Table~\hbox{B-8}`'中的例子说明过这个原理。另一个例子出现在输入参考文献的页码上: 在一行上只有 '22.' 不好看, 因此可以通过输入'`\hbox{13--22}.`'来禁止其断开'13–22'。另一方面, TEX 并不经常在连字处断行, 所以不必插入 `\hbox` 命令, 除非要修正 TEX 在前面处理中插入的不正确的断行。

▸ **练习14.1**
下面这些短语是从本手册(英文版)的前面各章精选出来的。想想作者是怎样输入它们的?

(cf. Chapter 12).
Chapters 12 and 21.
line 16 of Chapter 6's `story`
lines 7 to 11
lines 2, 3, 4, and 5.
(2) a big black bar
All 256 characters are initially of category 12,
letter `x` in family 1.
the factor f, where n is 1000 times f.

▸ **练习14.2**
怎样输入短语'for all n greater than n_0'?

▸ **练习14.3**
怎样输入'exercise 4.3.2–15'?

▸ **练习14.4**
为什么输入'`Chapter~12`'比输入'`\hbox{Chapter 12}`'好?


▸ **练习14.5**
TEX 有时候在数学公式的等号后面断行。怎样才能禁止计算机在公式'$x = 0$'中断行?

▶ **练习14.6**
想想怎样禁止 TEX 在出现的连字符和破折号后面断行。(这要用在长的参考书目中。)

有时候, 要允许在'/'后面断行, 就象它是一个连字符一样。为此, plain TEX 给出了'`\slash`'; 例如, '`input\slash output`'得到的就是'input/output', 并且中间可以断行。

如果要在段落中间的某个点强制断行, 只需要输入'`\break`'。但是, 这可能导致此行的字间距 太 大 了 。
如果希望 TEX 在强制断行前用空白把行右边的部分充满,
并且不在下一行缩进, 将使用'`\hfil\break`'。

如果连续输入的几行要照样逐行输出, 解决之道是在每行的结尾添加'`\par`'; 但有时比较麻烦, 因此 plain TEX 给出一个简写命令'`\obeylines`', 它的效果与每行后面输入 `\par` 是一样的。在使用 `\obeylines` 后, 每行就输出一行, 除非输入行以'`%`'结尾或太长必须断行。例如, 如果要输入诗, 可能要用到 `\obeylines`。一定要把 `\obeylines` 放在组中, 否则这种"诗歌模式"会延续到文档结束。

```
{\obeylines\smallskip
Roses are red,
\quad Violets are blue;
Rhymes can be typeset
\quad With boxes and glue.
\smallskip}
```

▶ **练习14.7**
解释一下这首诗中 `\quad` 的作用。如果所有'`\quad`'被'`\indent`'代替会出现什么情况?

严格讲, TEX 是用如下方法分段成行的: 断点要插入单词之间或连字符后面, 使得所得到的行的 badness 不超出当前的 `\tolerance`。如果无法这样插入断点, 就设置一个溢出的盒子。否则, 所选择的断点要使得段落在数学上是最优的, 即, 在所有可选择的断点序列中尽可能使缺陷最少。缺陷要考虑各个行的 badness, 连续两行都以连字符结束这种情况存在与否, 或者紧密的行后面跟着一个松散的行。

但是, 前一段落中断行的通俗描述比实际出现的东西太过简单了。本章剩下的部分就精确地讨论其细节, 这是为了在非标准方式下使用 TEX 的人们。已经证明, TEX 的断行算法一般足以处理惊人数量的不同类型的应用问题; 实际上这可能是 TEX 最有意义的方面。但是, 从现在开始的每段前面至少有一个危险标识, 所有可能要先学习初级阶段的内容, 而不是现在开始学下面的内容。

在分割成行之前, TEX 中的段落实际上是一个*水平列*, 即, TEX 在水平描述下集合起来的一系列项目。我们已经通俗地说过, 水平列由盒子和粘连组成; 其实盒子和粘连并不是所有的东西。水平列中的每个项目都是下列一种东西:

- 一个盒子(字符, 带子, 标尺, hbox, 或者 vbox);

- 一个任意可断点(马上就要解释);
- 一个“whatsit”(后面要讨论的某种特殊的东西);
- 垂直上的东西(来自 `\mark`, `\vadjust` 或者 `\insert`);
- 一个粘连团(或者 `\leaders`, 我们后面将会看到);
- 一个 kern (类似于粘连, 但是不能伸缩);
- 一个 penalty (用来表示此处断行的不适合度);
- “math-on”(公式开头)或者“math-off”(公式结尾)。

最后四项(粘连, kern, penalty 和数学项目)称为*可弃的*, 因为在断行处它们可能要改变或消失; 前四项称为不可弃的, 因为它们总是原封不动的。许多出现在水平列中的东西在本手册中都没有触及, 但是讨论断行也不需要学习它们。早晚你会知道上面所列的新东西要掺杂在水平列中; 而且如果要对 TEX 的整个内部过程有一个全面的认识, 总可以用 `\showlists` 以及各种语言特性来看看 TEX 正在干什么。

一个任意可断点由三个系列的字符组成, 叫做 *pre-break*, *post-break* 和 *no-break* 文本。 原理是, 如果可断点出现在此处, 那么 pre-break 文本出现在当前行的结尾, post-break 文本出现在下一行的开头; 但是如果没有出现可断点, 那么 no-break 文本将出现在当前行。通过给出

`\discretionary{`⟨pre-break text⟩`}{`⟨post-break text⟩`}{`⟨no-break text⟩`}`

用户可以完全普遍地给出任意可断点, 其中的三个文本完全由字符, 盒子和 kern 组成。 例如, 如果水平列包含,

```
di\discretionary{f-}{fi}{ffi}cult.
```

那么 TEX 可以在单词‘difficult’的‘f’之间添加连字符, 即使它使得‘ffi’这个组合字被分割成‘f-’和‘fi’这个组合字。幸好, 你不需要输入这些繁杂的东西; TEX 的连字算法在幕后运行, 在必要时会把组合字分开并在它们中间放上任意可断点。

最常用的任意可断点就是简单的任意连字

```
\discretionary{-}{}{}
```

对此 TEX 也允许省略写法‘`\-`’。下一个最常用的是

```
\discretionary{}{}{}
```

(一个“空任意可断点”), 这时 TEX 自动在‘`-`’后和每个以‘`-`’结束的组合字后插入任意可断点。对于 plain TEX, 空任意可断点会插入在连字符和破折号后。某种字体都有一个相应的 `\hyphenchar`, 我们可以直接把它假定为‘`-`’。)

当 TEX 把单词连字化时, 它直接把任意可断点插入到水平列中。例如, 如果需要连字化, 单词‘`discretionary hyphens`’就被转换成

```
dis\-cre\-tionary hy\-phens
```

但是 TEX 并不把连字算法应用到任何已经包含一个任意可断点的单词上; 因此, 在紧急情况下, 可以用明确给出的任意可断点来覆盖掉 TEX 自动生成的可断点。

▶ **练习14.8**

有些德语复合词当分割在两行时要改变它们的拼写。例如, 'backen'变成'bak-ken', 'Bettuch'变成'Bett-tuch'。怎样才能令 TEX 这样做?

为了节约时间, TEX 首先要在不插入任何连字符的情况下分段为行。如果所找到的一系列断点所得到的行的 badness 没有超出 `\pretolerance` 的值, 那么这第一次就通过了。如果第一次没通过, 就利用附录 H 的方法,通过把任意可断点插入水平列而把段落中的每个单词都连字化, 并且在第二次尝试时使用 `\tolerance` 而不是 `\pretolerance`。当行相当宽时, 比如象本手册一样, 实验表明, 90% 的时候都可以一次通过, 并且平均每段很少超过两个单词要用到连字算法。但是当行非常窄时, 第一次一般都很快失败了。Plain TEX 的默认值是 `\pretolerance=100` 和 `\tolerance=200`。如果 `\pretolerance=10000`, 第一次一般都顺利通过, 所以不会用到连字算法(并且间距可能宽得可怕); 另一方面, 如果 `\pretolerance=-1`, TEX 就跳过第一次, 直接使用连字。

断行只出现在水平列的某些地方。严格讲, 出现在单词之间和连字符后面, 但是实际上, 下列五种情况是允许的:

a) 在粘连处, 只要这个粘连后面跟的是一个不可弃项目, 并且它不是数学公式的一部分(即, 不在 math-on 和 math-off 之间)。"在粘连处"的断点出现在粘连间距的左边。

b) 在 kern 处, 只要这个 kern 后面直接跟的是粘连, 并且它不是数学公式的一部分。

c) 在 math-off 处, 其后紧接着的是粘连。

d) 在 penalty 处(在公式中, 它可以被自动插入)。

e) 在任意可断点。

注意, 如果两个粘连团依次出现, 那么第二个从不被选作断点, 因为它后面跟的是粘连(它是可弃的)。

每个潜在断点都有严格相应的"penalty", 它表示此处断点的"美学值"。对 (a), (b), (c), penalty 是零; 对 (d), 就是给出的 penalty; 对 (e), 如果 pre-break 非空, 那么 penalty 是 `\hyphenpenalty` 的当前值, 或者如果 pre-break 是空的, 那么就是 `\exhyphenpenalty` 的当前值。Plain TEX 设置 `\hyphenpenalty=50` 和 `\exhyphenpenalty=50`。

例如, 如果在段落的某处给出'`\penalty 100`', 那么此处就是断行的合理位置, 但是要符合 penalty 为 100。如果给出'`\penalty-100`', 就告诉了 TEX 这是断行的相当好的地方, 因为负的 penalty 实际上是一个"bonus(奉送品)"; 以 bonus 结尾的行可能还有"优点"(负的缺陷)。

任何大于等于 10000 的 penalty 都被看作大得不在此断行。在另一个极端下, 任何小于等于 −10000 的 penalty 都被看作小得总在此断行。plain TEX 的宏 `\nobreak` 的直接定义就是'`\penalty10000`', 因为它禁止在此处断行。Plain TEX 中的带子等价于'`\nobreak\␣`'; 在这种情况下, 在由 `\␣` 表示的粘连处不断行, 因为当粘连后面跟一个象 penalty 这样的可弃项目时, 它就不是一个合理的断点。

▶ **练习14.9**
想想看在 plain TEX 中宏 `\break` 的定义是什么。

▶ **练习14.10**
当给出 `\nobreak\break` 或 `\break\nobreak` 时, 将出现什么情况?

当断行实际出现时, TEX 将去掉此断点后的所有可弃项目, 直到遇见不可弃项目或者遇见另一个选定的断点。例如, 如果中间没有插入盒子, 那么一系列粘连和 penalty 将看作一个单位, 除非更好的断点系列包括了一个或多个 penalty。Math-on 和 math-off 其实是作为 kern 而给出由 `\mathsurround` 规定的间距; 这样的间距在公式出现的行首和行尾时将消失在断行中了, 这是源于上面罗列的规则。

行的 badness 是一个整数, 近似等于 100 乘以一个比值的立方, 这个比值就是为了得到所要求尺寸的 hbox, 行中的粘连必须伸缩的比例。例如, 如果行的总收缩能力是 10 points, 并且粘连被总共压缩了 9 points, 那么 badness 计算出的值是 73 (因为 $100 \times (9/10)^3 = 72.9$); 类似地, 行的伸长为其总伸长能力的两倍, 那么前 badness 为 800。但是如果这样得到的 badness 大于 10000, 那么就把它取为 10000。(参见第十二章中"粘连调整比例" r 和"粘连调整阶次" i 的讨论; 如果 $i \neq 0$, 那么伸缩能力是无限大, 因此 badness 是零, 否则 badness 近似等于 $\min(100r^3, 10000)$。) 溢出的盒子被看作无限糟糕; 所以要尽可能避免它们。

Badness 大于等于 13 的行的粘连调整比例超过 50%。如果其粘连要收缩, 我们将认为此行*太紧*, 如果要伸长, 此行就*松散*, 如果伸长时 badness 超过 100, 它就*空荡*了。但是, 如果 badness 小于等于 12, 就认为此行*适中*。如果相邻两行的分类却不相近, 即, 太紧的行后面跟着一个松散或空荡的行, 或者一个适中的行后面是一个空荡的行, 那么它们就称为*视觉不相容的*。

TEX 用各个行中出现的总*缺陷*来把每个可能的断点系列排序。目的是要选择总缺陷最少的断点系列。假定一个行的 badness 为 b, 再假定在行尾的相应于断点的 penalty 是 p。如上所述, 如果 $p \geq 10000$, 或者 b 超出当前的 tolerance 或 pretolerance, 那么 TEX 就不计及此行。否则, 此行的缺陷由下列公式进行计算:

$$d = \begin{cases} (l+b)^2 + p^2, & \text{if } 0 \leq p < 10000; \\ (l+b)^2 - p^2, & \text{if } -10000 < p < 0; \\ (l+b)^2, & \text{if } p \leq -10000. \end{cases}$$

这里的 l 是 `\linepenalty` 的当前值, 如果要把所有段落都压缩到最小的行数, 就可以增大此参数; plain TEX 设置 `\linepenalty=10`。例如, 如果 $l = 10$, 那么以粘连结尾的 badness 为 20 的行的缺陷为 $(10+20)^2 = 900$, 这是因为对粘连处的断点没有 penalty。把段落的总缺陷最小化粗略地讲就是把 badness 和 penalty 的平方和变成最小; 这通常意味着在整个断点系列中, 各个行的最大 badness 也是最小的。

▶ **练习14.11**
缺陷公式有一个奇怪的不连续性: 当 $p \leq -10000$ 时, 好像定义为 $d = (l+b)^2 - 10000^2$ 更好。你能给出这个明显矛盾的理由吗?

其它的缺陷来自相邻行的配对。如果两个紧接的行视觉不相容，如前面刚讨论过的那样，那么 \adjdemerits 的当前值被加到 d 上。如果两个紧接的行以任意可断点为结尾，那么要加上 \doublehyphendemerits。如果整段的倒数第二行以任意可断点结尾，就加上 \finalhyphendemerits。Plain TEX 的设置是 \adjdemerits=10000, \doublehyphendemerits=10000, \finalhyphendemerits=5000。缺陷的单位是"badness 的平方"，所以如果缺陷的因素有很多，那么把缺陷定性的参数就要相当大；但是，tolerance 和 penalty 与 badness 的单位是一样的。

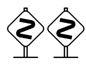
如果设置 \tracingparagraphs=1，那么 log 文件就包含了 TEX 断行运算的总结，所以当摆弄象 \linepenalty, \hyphenpenalty 和 \adjdemerits 这些参数时，就可以看到所出现的不同折衷方案。断行信息初看起来会有些不知所措，但是经过一点训练后将可以看懂了；实际上，这是真正理解断行的最好方法。下面是第六章的 story 的第二段的跟踪结果，其中 \hsize=2.5in 和 \tolerance=1000:

```
[]\tenrm Mr. Drofnats---or ‘‘R. J.,’’ as he pre-
@\discretionary via @@0 b=0 p=50 d=2600
@@1: line 1.2- t=2600 -> @@0
ferred to be called---was hap-pi-est when
@ via @@1 b=131 p=0 d=29881
@@2: line 2.0 t=32481 -> @@1
he
@ via @@1 b=25 p=0 d=1225
@@3: line 2.3 t=3825 -> @@1
was at work type-set-ting beau-ti-ful doc-
@\discretionary via @@2 b=1 p=50 d=12621
@\discretionary via @@3 b=291 p=50 d=103101
@@4: line 3.2- t=45102 -> @@2
u-
@\discretionary via @@3 b=44 p=50 d=15416
@@5: line 3.1- t=19241 -> @@3
ments.
@\par via @@4 b=0 p=-10000 d=5100
@\par via @@5 b=0 p=-10000 d=5100
@@6: line 4.2- t=24341 -> @@5
```

以'@@'开头的行表示*适宜的*断点，即，得到这些断点时没有任何超出 tolerance 的 badness。适宜的断点从 @@1 开始连续编号；段落的开头也被看作是适宜的，编号为 @@0。以'@'而不是以'@@'开头的行是接下来得到适宜断点的候选方法；当要做出选择时，TEX 将选择最好的候选方法。不以'@'开头的行表示 TEX 在本段进行到什么地方了。因此，例如，我们看到在'...hap-pi-est when'后面和'he'前面有'@@2: line 2.0 t=32481 -> @@1'，那么我们就知道了适宜断点 @@2 出现在单词 when 和 he 之间的空格上。符号'line 2.0'表示此适宜断点出现在第二行的结尾，并且此行是空荡。后缀 .0, .1, .2, .3 分别表示空荡，松散，适中和太

紧。) 如果行以任意可断点结尾, 或者此行是段落的最后一行, 那么在行号后面跟一个连字符; 例如, ‘line 1.2-’是连字化的一个适中的行。符号‘t=32481’表示从段落开头到 @@2 的总缺陷是 32481, 还有, ‘-> @@1’ 表示得到 @@2 的最好方法是来自 @@1。在跟踪信息的前一行, 我们看到计算行从 @@1 到此点的排版结果为: badness 为 131, penalty 为 0, 因此缺陷为 29881。类似地, 断点 @@3 给出了本段第二行的另一种方法, 其断点在‘he’和‘was’之间; 它得到的第二行太紧, 并且当加到第一行上时缺陷只是 3825, 所以看起来 @@3 比 @@2 更好。但是, 下一个适宜断点(@@4)出现在‘doc-’后面, 从 @@2 到 @@4 的行的缺陷只是 12621, 而从 @@3 到 @@4 的行的缺陷大到 103101; 因此, 从 @@0 到 @@4 的最好方法是通过 @@2。如果我们把缺陷看作距离, 那么 TeX 要找到从 @@0 到每个适宜断点的“最短路径”(所用的是在非循环图中求最短路径的著名算法的变形)。最后, 段落的结尾处在断点 @@6, 并且从 @@0 到 @@6 的最短路径代表了断点的最佳系列。从 @@6 沿着箭头往回, 我们就推导出在这个特殊段落中的最好断点是 @@5, @@3, @@1。

▶ **练习14.12**
解释一下为什么从 @@1 到 @@2 的缺陷是 29881, 从 @@2 到 @@4 的缺陷是 12621。


如果在这样的跟踪信息中出现了‘b=*’, 那么它表示必须选择一个不适宜的断点, 因为没有适宜的方法使得整体缺陷很小。


我们还没有讨论使段落的最后一行比其它行短的特殊技巧。就在 TeX 开始选择断点之前, 它要做两个很重要的事情: (1). 如果当前水平列的最后一个项目是粘连, 就扔掉此粘连。(原因是空格通常正好在 \par 之前或 $$ 之前进入记号列中, 并且此空格不被看作段落的一部分。) (2) 还有三个项目要放在当前水平列的结尾: \penalty10000(禁止断行); \hskip\parfillskip(把“最后的粘连”放在段中); \penalty-10000(强制最后的断行)。Plain TeX 设置 \parfillskip=0pt plus1fil, 因此每段的最后一行在需要时用空白充满; 但是在其它特殊应用中, 设置其它的 \parfillskip 也是应该的。例如, 当前段的结尾是右对齐, 因为它按照 \parfillskip=0pt 来排版; 这样的结果不需要修改什么, 因为长的段落一般可以有很大的弹性, 使得在所有地方都可以断行。 你可以在段落中玩些花样, 因为断行的算法通常好像是千里眼。只要输入足够长的段　　落　　即　　可　　。

▶ **练习14.13**
用户 Ben 要在段落结尾处添加上‘\hfilneg\par’, 其思路是, \hfilneg 的负伸缩性将与 plain TeX 的 \parfillskip 抵消。为什么这个聪明的想法不对?

▶ **练习14.14**
怎样设置 \parfillskip 才能使得段落的最后一行右边的空白与第一行的缩进一样大?

▶ **练习14.15**
因为 TeX 在确定断行之前要读入整个段, 所以, 如果要排版一段有 200 行哲学著作或现代小说, 计算机的内存可能会溢出。想个办法来应付它。


TeX 提供了 \leftskip 和 \rightskip 这两个参数, 用来给出段落中每行左右插入的粘连; 当计算 badness 和缺陷时, 要考虑这个粘连。Plain TeX 一般设置 \leftskip 和 \rightskip 为零, 但是有一个叫‘\narrower’的宏, 把它们的值增加为当前的 \parindent。当引用书中的大段内容时可能要用到

`\narrower`。

```
{\narrower\smallskip\noindent
This paragraph will have narrower lines than
the surrounding paragraphs do, because it
uses the ‘‘narrower’’ feature of plain \TeX.
The former margins will be restored after
this group ends.\smallskip}
```

(试试看。) 本例中的第二个‘`smallskip`’结束了本段。重要的是在结束组之前要结束段, 否则 `\narrower` 会在 TEX 确定断行之前失效。

▶ **练习14.16**
当用 italic 或 slanted 字体排版整个段落时, 有时候此段落会相对于其它段落在页面上有一个偏移。利用 `\leftskip` 和 `\rightskip` 把某段落的所有行向左移 1 pt。

▶ **练习14.17**
Plain TEX 中的宏 `\centerline`, `\leftline`, `\rightline` 和 `\line` 都没有考虑 `\leftskip` 和 `\rightskip`。怎样才能把它们算进去?

如果猜到了 `\raggedright` 就是用 `\rightskip` 做的处理, 那么你对了。但是有些要注意的。例如, 可以设置 `\rightskip=0pt plus1fil`, 并且每行的右边都用空白填满。但是, 这不是左对齐的很好的方法, 因为无限的伸长能力会把很短的行的 badness 变成零。为了让左对齐更好, 技巧是把 `\rightskip` 的伸长设置得可断行即可, 但不能太大, 这样短行就被看作糟糕的了。还有, 单词之间的间距应该固定, 让它们不要伸缩。(见附录 B 中 `\raggedright` 的定义。) 也还可以允许单词之间的粘连略微变化, 这样右边界不是那么太参差不齐, 而段落还是不齐的样子。

只有在断行时 TEX 才用到影响断行的参数。例如, 如果要 TEX 把这个词而不是那个词连字化, 那么不能在段落中间改变 `\hyphenpenalty`。`\hyphenpenalty`, `\rightskip`, `\hsize` 等等相应的值是它们在段尾的当前值。另一方面, 段落开头或你给出‘`\indent`’时得到的缩进宽度由 `\parindent` 的值确定, 在缩进读入当前水平列时就确定了, 而不是在段尾。类似地, 插入段落中的数学公式的 penalty 是在每个公式结尾处 `\binoppenalty` 和 `\relpenalty` 的当前值。附录 D 给出了一个例子, 不用 `\leftskip` 或 `\rightskip`, 而在一个段落中既有左对齐, 也有右对齐。

如果直接改变 `\leftskip` 和 `\rightskip` 还不能随心所欲, 就可以用更一般的方法来控制行的长度。例如, 为了给包含 Galileo 不朽名言的圆形图解留下地方, 当前段落被切掉一个半圆的洞; 本段和圆形区域的所有断行都遵循了 TEX 的断行算法。通过 `\parshape` =⟨number⟩, 你可以得到完全任意的段落形状, 其中 ⟨number⟩ 是正整数 n, 后面跟着 $2n$ ⟨dimen⟩ 参数。一般地, ‘`\parshape=`$n\ i_1\ l_1\ i_2\ l_2 \ldots i_n\ l_n$’ 给出这样的段落, 其前 n 行的长度分别为 $l_1, l_2, \ldots, l_n$, 并且在左边界缩进的量分别为 $i_1, i_2, \ldots, i_n$。如果段落的行不够 n 行, 剩下的

The area of a circle is a mean proportional between any two regular and similar polygons of which one circumscribes it and the other is isoperimetric with it. In addition, the area of the circle is less than that of any circumscribed polygon and greater than that of any isoperimetric polygon. And further, of these circumscribed polygons, the one that has the greater number of sides has a smaller area than the one that has a lesser number; but, on the other hand, the isoperimetric polygon that has the greater number of sides is the larger.
[Galileo, 1638]

参数将忽略掉; 如果超过 n 行, 第 n 行的参数将重复到最后。要取消前面
给 出 的 `\parshape` 用‘`\parshape=0`’ 。

▶ **练习14.18**
把下列 Pascal 的话排版在等腰三角形中: “I turn, in the following treatises, to various uses of those triangles whose generator is unity. But I leave out many more than I include; it is extraordinary how fertile in properties this triangle is. Everyone can try his hand.”

可能不常用到特殊的段落形状。但是, 有一个相当频繁出现的情况, 所以 TEX 为它提供了一种特殊的定义, 用两个参数 `\hangindent` 和 `\hangafter` 来表示。命令‘`\hangindent=`⟨dimen⟩’给出了所谓的悬挂缩进, 命令‘`\hangafter=`⟨number⟩’给出此缩进的范围。设 x 和 n 分别为 `\hangindent` 和 `\hangafter` 的值, h 是 `\hsize` 的值; 那么, 如果 $n \geq 0$, 悬挂缩进将出现在段落的第 $n+1$, $n+2$, ... 行, 但是如果 $n < 0$, 它就出现在第 $1, 2, \ldots, |n|$ 行。悬挂缩进的意思是, 行的宽度为 $h - |x|$, 而不是其正常宽度 h; 如果 $x \geq 0$, 行就在左边界缩进, 否则在右边界缩进。例如, 本手册的“危险”标识段就有一个 3 picas 的悬挂缩进, 持续 2 行; 它们的设置为 `\hangindent=3pc` 和 `\hangafter=-2`。

Plain TEX 在其‘`\item`’宏中用到悬挂缩进, 所得的段落中, 每行的缩进都是一个正常 `\indent`。还有, TEX 把一个参数放在第一行缩进的位置上。另一个叫 ‘`\itemitem`’ 的宏是一样的, 只是缩进了两个。例如, 假定输入的是

```
\item{1.} This is the first of several cases that are being
enumerated, with hanging indentation applied to entire paragraphs.
\itemitem{a)} This is the first subcase.
\itemitem{b)} And this is the second subcase. Notice
that subcases have twice as much hanging indentation.
\item{2.} The second case is similar.
```

那么就得到下面的输出结果:

1. This is the first of several cases that are being enumerated, with hanging indentation applied to entire paragraphs.
   a) This is the first subcase.
   b) And this is the second subcase. Notice that subcases have twice as much hanging indentation.
2. The second case is similar.

(Plain TEX 中的缩进实际上不如这里展示的好看; 附录 B 中‘`\parindent=20pt`’, 而本手册 `\parindent =36pt`。) 习惯上在编号的项目组前后要加上 `medskip`, 并且在对所有项目做最后讨论前要加 `\noindent`。在 `\item` 或 `\itemitem` 前不需要空行, 因为这些宏是以 `\par` 开头的。

▶ **练习14.19**
假定编号项目的长度为两段以上。 怎样使用 `\item` 才能在后续的段落中得到悬挂缩进?

▶ **练习14.20**

如果要用“•”代替“1.”，怎样做？

▶ **练习14.21**

宏‘\item’不能改变右边界。怎样才能在两边都缩进？

▶ **练习14.22**

怎样在第二行以后给出 -2 em 的悬挂缩进(即行要凸出左边界)？

如果 \parshape 和悬挂缩进同时给出，那么执行 \parshape 而忽略掉 \hangindent。所得到的是正常的段落形状，当 \parshape=0, \hangindent=0pt 和 \hangafter=1 时，其中每行的宽度都是 \hsize。TEX 在每个段落结束时自动复原这些正常值，并且(通过局部定义)随时进入内部垂直模式。例如，在 \vbox 构建之外出现的悬挂缩进不会出现在 vbox 内部，除非在内部给出它了。

▶ **练习14.23**

假定要在右边留下 15 行的一个长方形图示区，并且预料需要 3 个段落此图示区才够用。通过 TEX 重新安排悬挂缩进来设计一种好方法，不用调整即可完成此任务。

如果列表方程出现在非标准形状的段落中，TEX 总假定列表方程占据 3 行。例如，一个段落有 4 行文本，接着是一个列表方程，接着又是两行文本，那么其长度将被看作 $4+3+2=9$ 行；列表方程将按照第六行的段落形状来缩进和居中。

TEX 有一个内部整数变量叫做 \prevgraf，它记录了已完成或已部分完成的最近一个段落的行数。你可以把 \prevgraf 当做一个 ⟨number⟩ 使用，并且如果想要 TEX 认为自己是在当前段落形状的某些特殊地方，可以把 \prevgraf 设定为任意非负整数。例如，再次看看 4 行文本，一个列表公式，2 行文本的段落。当 TEX 开始此段落时，它设置 \prevgraf=0；当完成列表公式时，\prevgraf 是 7；并且在完成段落时是 9。如果列表公式实际上比通常要高一行，那么在最后两行开始前可以设置 \prevgraf=8；那么 TEX 将认为此段落为 10 行。只有当 TEX 处理非标准的 \parshape 或 \hangindent 时，\prevgraf 的值从起作用。

▶ **练习14.24**

利用 \prevgraf 重新解答 14.23。

现在你可能认为 TEX 的断行算法还有很多法宝，甚至更多。但是还只有一个，叫做“松散度(looseness)”；当要微调书的页数时，有一天可能要用到它。 如果设 \looseness=1，TEX 将试着把当前段落的行数比最佳行数增加一行，倘若可以在不超出所要求的各个行的 badness 的 tolerance 下选出这样的断点的话。类似地，如果设置 \looseness=2，TEX 就试着增加两行；如果 \looseness=-1，就试着变短。一般思路是，TEX 首先按照通常方法找到断点，接着如果最佳断点得到的行数是 n，并且如果 \looseness 是 1，为了在不超出当前 tolerance 的情况下尽可能使行数接近 $n+1$，TEX 将选择最后的断点。还有，最后的断点的总缺陷最少，再从剩下的得到同样数目的行。

例如，如果要避免在某些不好调整的页面上产生孤行，或者如果要在某些双栏文档中得到的行总数为偶数，可以设置 \looseness=1。通常最好选择已经相当“满”的段落，即最后一行

的空白比较少, 因为把这样的段落松散化没有不利的后果。可能还要在最后那个段落的两个单词之间添加带子, 这样才能避免在松散后的文档中最后一行出现孤词; 这个带子就去掉了你篡改间距的痕迹。另一方面, TEX 几乎可以把所有的长段落伸长一点而无害; 实际上, 当前段落就比其最佳长度多一行。

当 TEX 重新设置 `\hangindent`, `\hangafter` 和 `\parshape` 的同时, 就把松散度设置为零。

▶ **练习14.25**
如果设置 `\looseness=-1000` 会出现什么情况?

就在转到水平模式开始读入段落前, TEX 把由 `\parskip` 给出的粘连插入将包含此段落的垂直列中, 除非迄今为止此垂直列还是空的。例如, '`\parskip=3pt`'就把 3 points 的额外间距插入段落之间。Plain TEX 设置 `\parskip=0pt plus1pt`; 有一点伸长能力, 但是没有额外间距。

在断行完成后, TEX 把行追加到封装当前段落的当前垂直列中, 象第十而章讨论的那样插入行间粘连; 这个行间粘连来自当前起作用的 `\baselineskip`, `\lineskip` 和 `\lineskiplimit` 的值。为了有利于后面可能出现的控制断页, 就在每个行间粘连团前要把 penalty 插入垂直列。例如, 如果允许, 对在段落的前两行之间或最后一行之前的断页, 插入特殊的 penalty, 就可以避免在一个页面上出现与本段落其余行隔开的孤行。

下面是行间 penalty 的计算方法: TEX 选定某些段落或列表方程前面的段落部分的断点; 形成 n 行。在第 j 和 $j+1$ (j 的取值为 $1 \le j < n$)行之间的 penalty 等于 `\interlinepenalty` 的值加上特殊情况下的额外量: 如果 $j=1$, 即, 在第一行后, 加上 `\clubpenalty`; 接着如果 $j=n-1$, 即就在最后一行前, 根据当前行是否是列表方程的前一行, 加上 `\displaywidowpenalty` 或 `\widowpenalty`; 最后, 如果第 j 行的结尾是任意可断点, 加上 `\brokenpenalty`。Plain TEX 设置 `\clubpenalty=150`, `\widowpenalty=150`, `\displaywidowpenalty=50` 和 `\brokenpenalty=100`; `\interlinepenalty` 在正常时是零, 在脚注中增加到 100, 这样长的脚注就不会在页面之间断开。)

▶ **练习14.26**
看看一个 5 行的段落, 其中第二和第四行以连字符结尾。那么 plain TEX 将在行间放置的 penalty 是多少?

▶ **练习14.27**
在只有两行的段落中 penalty 是什么?

如果在段落中给出 `\vadjust{`⟨vertical list⟩`}`, 那么 TEX 将把给出的内部垂直列插入到封装本段落的垂直列中, 直接插入在包含 `\vadjust` 的位置的任意行后面。例如, 如果行看起来太紧, 使用'`\vadjust{\kern1pt}`'就可以增加本段落的行间距。(为了说明它, 作者在前一行使用了它。) 还有, 如果要确保断页在某行后面出现, 就可以在此行的任意处使用'`\vadjust{\eject}`'。

后面的章节讨论了命令 `\insert` 和 `\mark`, 它们与 TEX 的页面构建有关。如果这样的命令出现在段落中, 它们将会从包含它们的水平列中提出来, 与可能出现的来自命令 `\vadjust` 的其它垂直指

令放在封装的垂直列中。在最后的垂直列中, 文本的每个水平列是有关 hbox, 它在行间粘连后面, 在从行中提出来的垂直指令前面(如果有几个垂直指令, 要保持从左到右的顺序); 接下来是行间 penalty, 如果它不等于零。插入的垂直指令不影响行间粘连。

▶ **练习14.28**
设计一个叫 `\marginalstar` 的宏, 可以在段落中任何地方使用。当 `\marginalstar` 出现时, 它在行的左边界内侧放一个星号。

当 TeX 进入水平模式时, 将中断由命令 `\everypar={`⟨token list⟩`}` 预定义的正常记号读入。例如, 假定已经给出'`\everypar={A}`'。如果在垂直模式输入'`B`', 那么 TeX 将转到水平模式(在把粘连 `\parskip` 放置在当前页面以后), 并且通过插入宽度为 `\parindent` 的空盒子来开始一个水平列。接着, TeX 将读入'`AB`', 因为它在回来得到'`B`'之前读入了记号 `\everypar`, 而'`B`'引发了一个新段。当然, 这不是 `\everypar` 很好的示例; 但是如果你发挥想像, 就能找到更好应用。

▶ **练习14.29**
利用 `\everypar` 定义一个宏 `\insertbullets`: 在形式'`{\insertbullets ...\par}`'中的所有段落都把小黑点符号'•'放在要缩进的地方。

如果给出'`\noindent\par`', 就得到零行的段落。如果 `\everypar` 是空的, 这样的段落在当前垂直列中除了 `\parskip` 粘连外什么也没有。

▶ **练习14.30**
猜猜看, 如果给出'`\noindent$$...$$ \par`'会出现什么?

经验表明, TeX 的断行算法可以处理的任务类型多得惊人。例如, 下面是一种可能的应用示例: 在*Mathematical Reviews*出版的文章一般要署上作者的名字和地址, 并且这些内容要居右排版, 即在右边界。如果在段落的最后一行的右边有足够的空间来放名字和地址, 那么出版社就可以节约空间, 并且同时因为页面上没有奇怪间隙所以输出结果也好。

This is a case where the name and address fit in nicely
with the review. *A. Reviewer* (Ann Arbor, Mich.)

But sometimes an extra line must be added.
N. Bourbaki (Paris)

假设如果在文章和作者在同一行, 那么它们之间至少要有两个 em 的间距。我们将设计一个宏, 使得如下排版时得到上面的例子:

```
... with the review. \signed A. Reviewer (Ann Arbor, Mich.)
... an extra line must be added. \signed N. Bourbaki (Paris)
```

宏的定义为

```
\def\signed #1 (#2){{\unskip\nobreak\hfil\penalty50
  \hskip2em\hbox{}\nobreak\hfil\sl#1\/ \rm(#2)
  \parfillskip=0pt \finalhyphendemerits=0 \par}}
```

如果断行出现在 `\penalty50`, 那么 `\hskip2em` 就不使用, 并且在行的开头出现一个空 `\hbox`, 后面是 `\hfil` 粘连。它得到 badness 为零的两个行; 其中第一个行的 penalty 为 50。但是如果断行不出现在 `\penalty50`, 那么就在文章和作者之间出现一个 2 em plus 2 fil 的粘连; 它得到的是 badness 为零的一个行。 TEX 将在这二者之间选择总缺陷最少的。如果可以, 单行的结果右边最好。

▶ 练习14.31
如果在 `\signed` 宏中把'`\hbox{}`'去掉会出现什么情况?

▶ 练习14.32
在 `\signed` 宏中为什么有'`\finalhyphendemerits=0`'?

▶ 练习14.33
在本章前面的某段中, 作者在一个特定的地方用 `\break` 强制断行; 因此, 此段的第三行的确散开了。
为什么所有额外的间距都放在第三行而不是均匀地放在前三行?

如果要不惜代价地自动去掉溢出的盒子, 可以试着设置 `tolerance=10000`; 在难处理时它容许任意糟糕的行。但是无限的 tolerance 是一个坏主意, 因为 TEX 不再区分糟糕和荒谬的行。的确, tolerance 为 10000 时就助长 TEX 把所有的 badness 集中在一个地方, 用一个实在难看的行来代替两个不太难看的行, 因为按照规则, 单个"牺牲"得到的总缺陷最少。还有得到所要求结果的更好方法: TEX 有一个叫 `\emergencystretch` 的参数, 如果溢出的盒子用别的方法无法避免, 那么在计算 badness 和总缺陷时就把它加到每行设定的输出能力上。如果 `\emergencystretch` 是正的, 当第一种途径无法满足 `\pretolerance` 和 `\tolerance` 时, TEX 就在选择断行前在此段落上产生第三条途径。`\emergencystretch` 的作用是按比例缩小 badness, 使得大无限与小无限的区别保留下来。通过把 `\emergencystretch` 设置得足够高(根据 `\hsize`), 可以确保不超出 `\tolerance`; 因此, 溢出的盒子就再也出现不了, 除非实在无法断行。

▶ 练习14.34
设计一个宏 `\raggedcenter` (类似于 `\raggedright`), 把段落中的单词分为尽可能少的等长的行, 并且把行居中。尽可能避免出现连字符。

15. 组行为页

TEX 尝试着在适当的位置把文档分为各个页面, 并且其此项技术一般效果很好。但是, 构建页面的问题比我们前一章讨论的断行问题更难, 因为页面一般比行的伸缩性更小。如果一个页面的垂直粘连只有很少或没有伸缩性, 那么 TEX 一般只能按部就班地换页; 反过来, 如果粘连上有很多变化, 又会因为不同页面之间不一致而使得结果很糟糕。因此, 如果你对页面的外观比较挑剔, 就要做一些修改, 直到得到满意的效果为止, 或者可能要用到第十四章讨论过的 `\looseness`; 没有比你更自动的方法来完成此任务了。

有很多列表公式的数学文章在这方面有优势, 因为列表公式周围的粘连比较灵活。当在某些段落之间使用 `\smallskip`, `\medskip` 或 `\bigskip` 间距时, TEX 还得到可机动的宝贵空间。例如, 看看包含大约一打练习的页面, 并且设定练习之间有 3 pt 的额外间距, 其中此间距可伸长 4 pt 或收缩 2 pt。那么, 为了避免把一个练习分在两个页面上, 就可以在一个页面上挤出一行或去掉一行。类似地, 在会员花名册或公司电话簿等特殊出版物中, 可以灵活使用粘连, 使得各个单个单位不需要分在两栏或两页上, 而且每栏的高度还一致。

对普通用户而言, TEX 的自动分页方法就满足了。并且当所得到的是不好的结果时, 可以通过‘`\eject`’强制 TEX 在所要求的地方分页。但是要注意, 如果需要的话, `\eject` 将使 TEX 伸长以充满页面, 使得此页面的底部和顶部基线与其它页面一致。如果要 eject 得到一个短的页面, 在底部用空白把它充满, 就要用‘`\vfill\eject`’。

如果在段落中间给出‘`\eject`’, 段落将首先被结束, 就象键入‘`\par\eject`’一样。但在第十四章提到, 如果要在整个段落断行完毕后在包含当前位置的行后面强制分页, 可以在段落中间使用‘`\vadjust {\eject}`’; 此段剩下的行将放在下一页。

为了禁止分页, 可以在垂直模式下输入‘`\nobreak`’, 就象在水平模式下用 `\nobreak` 禁止断行一样。例如, 在小节的题目和它的第一行文本之间加上 `\nobreak` 是明智之举。但是 `\nobreak` 不能抵消掉象 `\eject` 这样的其它分页命令; 它只禁止在后面紧跟的粘连处分页。如果要有效地控制所有东西, 就必须熟悉 TEX 的断行和分页规则。本章剩下的内容就来讨论分页的细节。

TEX 通过计算 badness 率和 penalty 把行的列表分页, 这或多或少类似于分段成行。但是一次只生成一个页面, 并且完成后从 TEX 的内存中要清除掉; 不能往前看看本分页怎样影响前一个页面。换句话说, TEX 用一种特殊的方法在一整段中找出行的最佳断点, 但是不能在整个文档中找到页面的最佳断点。计算机没有足够的高速内存来存储好几页的内容, 所有 TEX 直接通过一种“局部”方法来尽力选择每个页面得到, 而不是用“整体”优化方法。

现在来看看 TEX 构建页面的细节。所有输入到文档页面东西都放在*主垂直列*中, 它是 TEX 在垂直模式下积累起来的项目序列。在垂直列中的每个项目是下列某种东西:

- 一个盒子(一个 hbox 或 vbox 或标尺);
- 一个“whatsit”(后面要解释的特殊东西);
- 一个标记(后面要解释的另一种东西);
- 一个插入(仍然是我们将得到的另一种东西);

- 一个粘连团(或者 \leaders, 我们后面将会看到);
- 一个 kern (象粘连, 但不能伸缩);
- 一个 penalty (表示此处分页的不良度)

最后三种(粘连, kern 和 penalty 项)称为可弃的, 理由与水平列中的可弃性一样。如果要把这些说明与第十四章中水平情况下类似的规则作比较; 就发现垂直列很象水平列, 除了字符盒子, 任意可断点, \vadjust 项和数学转换不能出现在垂直列之外。第十二章列出了用 TEX 的内部盒子和粘连表示的一个典型垂直列。


分页只能出现在垂直列中的某个地方。合法的断点与水平情况下正好一样, 即:

a) 在粘连处, 如果此粘连直接在一个非可弃项目(即, 盒子, whatsit, 标记或者插入)前面;

b) 在 kern 处, 如果此 kern 直接跟着粘连;

c) 在 penalty 处(它可能已经直接被插入到段落中了)。

就象第十二章中讨论的那样, 行间粘连一般自动插入到垂直列的盒子之间, 所以在盒子之间一般有一个可用的断点。


就象在水平列中一样, 某个可能的断点有一个相应的 penalty, 对不好的断点它的值就高, 对好的断点就是负的。在粘连和 kern 处的断点的 penalty 是零, 所以只有在明确的 penalty 处的断点是非零的。如果在两个段落之间给出'\penalty-100', 就告诉了 TEX 应该在此处分页, 因为 penalty 是负值; 此处断点的 100 点积分本质上可以抵消必须在此处分页时的 100 单位的 badness。10000 以上的 penalty 大得足以禁止分页; 小于 -10000 的 penalty 小到足以强制分页。


Plain TEX 提供了几个控制分页的控制系列。例如, \smallbreak, \medbreak 和 \bigbreak 给出了越来越强的分页倾向, 其 penalty 分别为 -50, -100, -200; 还有, 如果它没有导致分页, 就分别插入 \smallskip, \medskip 或 \bigskip的行间距。但是, \smallbreak, \medbreak 和 \bigbreak 不在原有的粘连上添加不必要的粘连; 例如, 如果在列表方程后面添加 \samllbreak, 那么除了来自列表方程的粘连外, 不会得到 \smallskip 的行间距。因此, 在数学论文中, 这些命令可以很方便地在定理的陈述前后使用。在本手册中, 作者在每个危险标识段落前后都放了 \medbreak; \medbreak\medbreak 等价于一个 \medbreak, 所以当这样的段落在一个结束而另一个接着开始时不会得到两个 medskip。


宏 \goodbreak 的定义为'\par\penalty-500'。在校对时, 如果希望把某些页面略微伸长一点来改善其后的页面排版, 把它插入到文稿中就可以了。稍后, 如果改变后这个 \goodbreak 命令并不出现在页面底部, 那么它就不起作用了; 因此它不如 eject 强制性大。


Plain TEX 提供的构建页面的最有效的宏叫做 \filbreak。严格讲, 它的意思是, "在此处分页, 并且用空白填满到底部, 除非下面还可以再放下带 \filbreak 的下一个对象"。因此, 如果在每个段落的结尾都放上 \filbreak, 并且你的段落不是太长, 那么分页正好出现在段落之间, 并且 TEX 在每个页面放尽可能多的段落。按照附录 B, \filbreak 的确切意思是:

```
\vfil\penalty-200\vfilneg
```

并且这个 TEX 原始控制系列的简单组合得到了所要的结果: 如果在 `\penalty-200` 处取了断点, 那么前面的 `\vfil` 就用空白填满到底部, 并且断点后的 `\vfilneg` 将被忽略; 但是如果没有在 penalty 处取断点, 那么 `\vfil` 和 `\vfilneg` 将互相抵消而没有任何影响。

Plain TEX 还提供了一个命令叫 `\raggedbottom`, 它类似于 `\raggedright`: 允许 TEX 在不同页面留不同的小空白, 以保证其它的行间距是一致的。

在第十四章我们看到, 段落的断点是通过计算每行的"缺陷"并把所有行的缺陷加起来而确定的。页面的方法更简单, 因为每个页面单独来考虑。通过下列公式 TEX 计算出分页的"成本":

$$c = \begin{cases} p, & \text{if } b < \infty \text{ and } p \le -10000 \text{ and } q < 10000; \\ b + p + q, & \text{if } b < 10000 \text{ and } -10000 < p < 10000 \text{ and } q < 10000; \\ 100000, & \text{if } b = 10000 \text{ and } -10000 < p < 10000 \text{ and } q < 10000; \\ \infty, & \text{if } (b = \infty \text{ or } q \ge 10000) \text{ and } p < 10000. \end{cases}$$

其中 b 是此处分页所生成页面的 badness; p 是当前断点的相应 penalty; q 是'`\insertpenalties`', 即页面上把插入盒子分开时的所有 penalty 的和, 其解释在后面。垂直 badness 的计算规则与水平 badness 一样; 它是 0 和 10000 这个范围的整数, 但是当盒子溢出时它是 ∞(无限大)。

当完成一个页面时, 把它从主垂直列中清除并且送到"输出例行程序", 这个我们将在后面看到; 所以其盒子和粘连最后都从 TEX 的内存中清除了。主垂直列剩下的东西存在在两个部分中: 首先是来自"当前页面", 它包含迄今为止 TEX 为分出下一页所考虑的所有内容; 接着是"备选内容", 即一旦 TEX 发现可行就放入当前页面的那些项目。如果使用 `\showlists`, TEX 将显示当前页面的内容和备选内容, 或者放在 log 文件中。(第十三章的例子没有给出这些列表, 因为在那种情况下填满都是空的。第二十四章更多地讨论了 TEX 的时机选择。)

只要 TEX 把一个项目从"备选内容"的顶部移到"当前页面"的底部, 那么如果当前页面不包含任何盒子, 那么它就忽略掉可弃项目(粘连, kern 或 penalty)。这就是粘连在分页处消失的方法。否则, 如果可弃项目是一个合理断点, 那么 TEX 就用前面上面给出的公式计算在此断点的成本 c。如果所得到的 c 小于等于迄今为止当前页面的其它成本, 那么 TEX 就把当前断点记为迄今为止最佳断点。并且如果 $c = \infty$ 或者 $p \le -10000$, TEX 就主动出击, 在所记住的最佳断点处分页。当前页面上最佳断点后面的任何东西都移回备选内容列, 它们会被再次处理; 因此, 在断点选定前"当前页面"一般得到比一个页面更多的内容。

在你看到这个程序运转前, 可能比较神秘。幸运的是, 有一个便利的方法来观察它; 你可以使用 `\tracingpages=1`, 从而告诉 TEX 把页面成本的计算输出到 log 文件中。例如, 当在本章的开头使

用 \tracingpages=1 时, 下面的内容就出现在 log 文件中:

```
%% goal height=528.0, max depth=2.2
% t=10.0 g=528.0 b=10000 p=150 c=100000#
% t=22.0 g=528.0 b=10000 p=0 c=100000#
% t=34.0 g=528.0 b=10000 p=0 c=100000#
    ⋮  (25 similar lines are being omitted here)
% t=346.0 plus 2.0 g=528.0 b=10000 p=0 c=100000#
% t=358.0 plus 2.0 g=528.0 b=10000 p=150 c=100000#
% t=370.02223 plus 2.0 g=528.0 b=10000 p=-100 c=100000#
% t=398.0 plus 5.0 minus 2.0 g=528.0 b=10000 p=0 c=100000#
% t=409.0 plus 5.0 minus 2.0 g=528.0 b=10000 p=0 c=100000#
% t=420.0 plus 5.0 minus 2.0 g=528.0 b=10000 p=150 c=100000#
% t=431.0 plus 5.0 minus 2.0 g=528.0 b=10000 p=-100 c=100000#
% t=459.0 plus 8.0 minus 4.0 g=528.0 b=10000 p=0 c=100000#
% t=470.0 plus 8.0 minus 4.0 g=528.0 b=10000 p=0 c=100000#
% t=481.0 plus 8.0 minus 4.0 g=528.0 b=10000 p=0 c=100000#
% t=492.0 plus 8.0 minus 4.0 g=528.0 b=10000 p=0 c=100000#
% t=503.0 plus 8.0 minus 4.0 g=528.0 b=3049 p=0 c=3049#
% t=514.0 plus 8.0 minus 4.0 g=528.0 b=533 p=150 c=683#
% t=525.0 plus 8.0 minus 4.0 g=528.0 b=5 p=-100 c=-95#
% t=553.0 plus 11.0 minus 6.0 g=528.0 b=* p=0 c=*
```

这个跟踪输出不可否认在外观上不那么“友好”, 但是毕竟它来自 TEX 的内脏中, 所做的事情已经简化为数字计算了。经过一些训练就可以读懂它们, 不过你不必经常这样做, 除非要做特殊的分页。其意思如下: 第一行以‘%%’开头, 当第一个盒子或插入进入当前页面列时写入; 它显示了此页面将要使用的“目标高度”和“最大深度”(即 \vsize 和 \maxdepth 的当前值)。在本手册中, \vsize=44pc 和 \maxdepth=2.2pt; log 文件中尺寸的单位是 point。随后的行以单个‘%’开头, 只要从备选内容向当前页面列移入合理的断点, 就写入。每个 % 行显示 t (如果在此分页, 它就是迄今为止的总高度)和 g (它是目标高度); 在本例中, g 保持固定, 为 528 pt, 但是如果诸如脚注这样的插入出现在页面上, g 应该减小。 t 的值稳定增大, 从 10 到 22 到 34 等等; 在页面顶部基线的间隔为 12 pt, 在底部为 11 pt(因为文本是 9 point 的字体)。只要一个文本盒子放在当前页面上, 就看到一个 % 行。但是, % 行也可以由 hbox 后的 penalty 或粘连项目生成, 而不是盒子自身。每个 % 行还显示了与断点相应的 badness b, penalty p, 和成本 c; 如果这个成本是迄今为止最好的, 就用‘#’来标记, 意思是, “如果后面没有更好的就把此断点应用于当前页面”。注意, 大约前 40 个断点都是 $b = 10000$, 这是因为它们糟糕得 TEX 认为无法区分; 在这种情况下 $c = 100000$, 所以 TEX 直接积累材料, 直到页面满到 $b < 10000$。penalty 为 150 反映了 \clubpenalty 或 \widowpenalty 被插入, 就象第十四章中讨论的那样。含有 p=-100 的三个行是“危险”标识段落之间的断点; 它们来自命令 \medbreak。在最后一行的符号 b=* 和 c=* 意味着 b 和 c 是无限大; 总高度 553 pt 不能通过收缩可用的粘连而减小到 528 pt。因此, 页面在

最佳的前一个位置结束, 这就是一个机动好断点: b=5 和 p=-100 断点净成本为 -95。

▶ **练习15.1**
假定例页的底部的段落有一个更短的行; 分页会怎样选定呢?

▶ **练习15.2**
在本例的最后两个“% 行”显示出自然高度跳过了 28 pt, 从 525.0 到 553.0。看看为什么有这么大的跳跃?

如果页面上底部盒子太深, 参数 \maxdepth 告诉 TeX 升高此盒子, 使得所构建页面的深度不超出所给值。(参见第十二章中 \boxmaxdepth 的讨论。) 在我们的例子中, \maxdepth=2.2pt, 并且此参数的影响可以在‘% t=370.02223’行看到。一般地, 在此断点 t 应该是 370.0; 但是在它前面的盒子是不寻常的, 因为它包含了 \tt 中的字母 j, 并且应该 10-point 的 typewriter 字体的 j 比基线低 2.22223 pt。因此, TeX 按照 hbox 的高度增加 .02223 pt 和深度仅仅为 2.2 pt 来表示 badness。

注意, 例子中的第一个“% 行给出了 t=10.0; 这是另一个参数的结果, 它叫做 \topskip。粘连在分页时消失了, 但是只要可以, 就希望所得页面的顶基线和底基线出现在预先确定的位置上; 从而 TeX 在每页的第一个盒子前面插入特殊粘连。这个特殊粘连等于 \topskip, 但是自然间距已经被第一个盒子的高度减小或在负值的情况下它被设置为零除外。例如, 如果 \topskip=20pt plus2pt, 并且当前页面上第一个盒子的高度为 13 pt, 那么 TeX 仅在盒子上方插入‘\vskip7pt plus2pt’。还有, 如果第一个盒子的高度超过 20 pt, 那么插入‘\vskip0pt plus2pt’。但是这个例子不是典型的, 因为一般 \topskip 没有伸缩性; plain TeX 设置 \topskip=10pt。

▶ **练习15.3**
假定 \vsize=528pt, \maxdepth=2.2pt, \topskip=10pt, 并且不使用 \insert 命令。 TeX 将构建高度为 528 pt 的页面, 并且下列两个陈述一般是正确的: (a). 页面最顶部的基线距离顶部 10 pt, 即, 比页面自己的基线高 518 pt。(b). 页面顶部的基线与页面自己的基线重合。看看在什么情况下 (a) 和 (b) 失效。

因为 \vsize, \maxdepth 和 \topskip 是参数, 所以可以在任何时候改变它们; 如果这样会出现什么结果? 嗯, 当 TeX 输出“%% 行”时, 即当第一个盒子或插入出现在当前页面时, 它会记住 \vsize 和 \maxdepth 的值; 直到下一页开始对这两个参数的改变才起作用。另一方面, 只有当第一个盒子送到当前页面时 TeX 才查看 \topskip 的值。如果插入出现在盒子前面, 那么此盒子之前的 \topskip 粘连就被看作一个可用断点; 这是整个页面不可能包含盒子的唯一情形。

在分页中所看到的 t 和 g 的值分别指 ⟨dimen⟩ 值‘\pagetotal’和‘\pagegoal’。你也可以改变它们(但是我们希望你知道怎样做)。例如, 命令 \pagegoal=500pt 将取代前面保存的 \vsize 的值。\pagetotal 表示累计的自然高度, 除了它之外, TeX 还有其它量 \pagestretch, \pagefilstretch, \pagefillstretch, \pagefilllstretch, \pageshrink 和 \pagedepth。如果当前页不包含盒子, 那么 \pagetotal 及与其相关的量等于零, 并且 \pagegoal 为 16383.99998 pt(TeX 的最大尺寸); 在这种情况下改变它们的值没有影响。式子中表示页面成本的整数 q 也可用在检验中, 并且是可改变的; 它的名称为 \insertpenalties。

值得一提的是, 分页与断行在一个小的方面是不同的: 如果使用 `\eject\eject`, 那么第二个 `\eject` 将被忽略, 因为它等价于 `\penalty-10000`, 而分页后的 penalty 是可弃的。但是如果在段落中使用 `\break\break`, 那么第二个 `\break` 产生一个空行, 因为只有当 penalty 不属于最后的断点序列时, 段落中断点后的 penalty 才是可弃的。实际上这个技术要点并不重要, 因为 `\break\break` 不是产生空行的好办法; 因为它只包含 `\leftskip` 和 `\rightskip` 这两个粘连, 所以得到的是松散的 hbox。类似地, '`\eject\eject`'不是一个生成空页的好方法, 即使 TEX 改变一下规则使得 `\eject` 不被忽略掉。得到空页的最好方法是'`\eject\line{}\vfil\eject`', 而生成空行的最好方法是'`\break\hbox{}\hfil\break`'。这两种方法都避免出现松散的盒子。

可能你想知道页码和此类内容是怎样添加到页面上的。答案是, 在每个分页选定之后, TEX 允许你做进一步处理; 在页面得到它们的最终格式前, 进入一个特殊的"输出例行程序"。第二十三章讨论了怎样构建输出例行程序以及怎样修改 plain TEX 的输出例行程序。

TEX 偶尔会生成一个难看的页面, 想知道出现什么问题吗? 例如, 你得到的可能只是一个段落和很多空白, 而下一页的文本空页却明显能放在这些空白处。这样明显反常的原因几乎都是无法找到好的断点; 即使对你而言另一种方法很好, 而对 TEX 而言却很糟! 因为 TEX 无法区分两个大于等于 10000 的 badness, 即使一个看起来比另一个很糟。在此情况下, 解决的方法是在某些可接受的地方插入 `\eject` 或者 `\vfill\eject`, 或者修订你的文稿。但是如果这个问题经常出现, 那么说明你使用的页面格式限制得太严格; 试着看看 `\tracingpages` 的输出信息, 并且修改一些 TEX 的参数, 这样就可以感觉好一些了。

本章剩下的内容将讨论插入对象: 就象脚注和图例这些东西, 以及它们与分页之间的互相影响。在我们讨论 TEX 处理插入对象的原始命令之前, 先看看 plain TEX 提供的高级工具。

在 plain TEX 中有几种方法可以插入图例。最简单的叫做"顶部浮动插入"; 命令是

`\topinsert`⟨vertical mode material⟩`\endinsert`

并且 TEX 试着把垂直模式的内容放在当前页面顶部。如果在某个页面顶部没有空间了, TEX 将把它插入到下一个页面顶部。⟨vertical mode material⟩ 可包含嵌入的段落(它按通常的方法临时中断垂直模式而得到); 例如:

```
\topinsert \vskip 2in
\hsize=3in \raggedright
\noindent{\bf Figure 3.} This is the caption to the
third illustration of my paper. I have left two inches
of space above the caption so that there will be room
to introduce special artwork. \endinsert
```

本例的 caption 设置放在页面左边, 栏宽为 3 inch, 左对齐。Plain TEX 自动在每个顶部插入下面添加一个"bigskip"; 它就把标题和文字分开。因为暗含了编组命令, 所以 `\hsize=3in` 和 `\raggedright` 的作用不会超出 `\endinsert`。

▶ **练习15.4**
修改本例, 使得下一个标题出现在右页边, 而不是左页边。

类似地, 如果使用'`\pageinsert`⟨vertical mode material⟩`\endinsert`', 那么垂直模式的内容将被调整为整个页面的大小(它下面没有 bigskip 了); 所得到的内容将放在下一页上。

还有'`\midinsert`⟨vertical mode material⟩`\endinsert`', 它首先把此内容插入其所在的位置, 在当前页面中间。如果没有足够的空间, 就得到了

`\bigskip\vbox{`⟨vertical mode material⟩`}\bigbreak`

否则, `\midinsert` 就转换为 `\topinsert`。偶尔 `\midinsert` 无法找到最佳位置, 因为 TEX 有时候先于当前页面来处理文本。可能要在 `\midinsert` 前加上'`\goodbreak`'。

应该在垂直模式下(即段落之间)使用命令 `\topinsert`, `\pageinsert`, `\midinsert`, 而不能在盒子或其它插入对象中。

如果紧接着有两个或多个 `\topinsert` 或 `\pageinsert` 命令, 那么 TEX 可能需要把它们顺次放在几个连续的页面上; 但是当转移时要保留其相对顺序。例如, 假定页面的高度为 9 inch, 而且在某些页面上已经放了 4 inch 的文本, 比如第 25 页。那么, 如果一行中有 7 个顶部插入对象, 大小分别为 1, 2, 3, 9, 3, 2, 1 inch; 9 inch 的其实就是 `\pageinsert`。会出现什么情况? 嗯, 第一, 第二个出现在第 25 页顶部, 接下来是已经输入的 4 inch 文本; 再接下来是你在 7 个插入后面输入的文本, 有 2 inch。第三个顶部插入出现在第 26 页顶部, 接下来是 6 inch 文本; 第四个满满地放在第 27 页; 剩下的三个放在第 28 页。

▶ **练习15.5**
在上面讨论的例子中, 如果最后一个 1 inch 的插入对象是中间插入而不是顶部插入会出现什么结果?

在论文的结尾, 可能希望确保没有把插入对象丢失掉; 在章的结尾, 可能要确保插入对象不要浮动到下一章。如果使用'`\vfill\supereject`', 那么 plain TEX 将把所有剩下的插入对象排版, 用空白把未填满的页面充满。

除了在页面顶部插入图例外, plain TEX 还在页面底部插入脚注。在段落中提供了 `\footnote` 宏来完成任务; * 例如, 在本句中的脚注是如下输入的:

```
... paragraphs;\footnote*{Like this.} for example, ...
```

一个 `\footnote` 有两个参数; 第一个是参照标记, 它既出现在段落中**, 也出现在脚注自身中, 接下来是脚注文字。[45] 后面的文本可以有几段长, 并且可以包含列表方程和此类对象, 但是不应包含其它插入对象。

---

* Like this.

** The author typed '`paragraph\footnote{**}{The author ...}`' here.

[45] And '`footnote.\footnote{$^{45}$}{And ...}`' here. The footnotes in this manual appear in smaller type, and they are set with hanging indentation; furthermore a smallskip occurs between footnotes on the same page. But in plain TEX, footnotes are typeset with the normal size of type, with `\textindent` used for the reference mark, and without extra smallskips. The `\textindent` macro is like `\item`, but it omits hanging indentation.

TEX 会确保每个脚注出现在其参照标记所在页面的底部。† 在必要时, 长的脚注会被分成两部分, 并且续在下一页底部, 就象你看到的下面的人为的例子一样。要思路流畅的话应尽可能避免使用脚注, 因为脚注会分散注意力。‡

宏 `\footnote` 应该只在段落或出现在 TEX 主垂直列的 hbox 中使用; 如果它们出现在盒子中的盒子中, 那么插入对象将丢失。因此, 例如, 不要把 `\footnote` 放在数学公式的子式中。但是, 在 `\centerline` 中使用脚注是可以的, 比如,

```
\centerline{A paper by A. U. Thor%
  \footnote*{Supported by NSF.}}
```

或者甚至在 `\halign` 中表格单元的外层使用。

只有顶部插入的话可以顺利处理, 只有脚注的话也行, 但是如果用不常用的方法把它们混合起来就复杂了。例如, 如果 `\pageinsert` 浮动在页面上, 而下面有必须要分开的脚注, 两个要同步的插入对象可能都要把自己放在同一页面上, 并且得到应该溢出的页面。还有, 插入对象不能出现在插入对象中, 所以不能在 `\topinsert` 中使用 `\footnote`。如果的确在某些标题上需要一个脚注, 那么有一个在垂直模式下使用的宏 `\vfootnote`。在使用时, 把象'`*`'这样的参照标记放在标题上, 并且把'`\vfootnote*{The footnote}`'放在标题最后可能出现的页面上的某处。在这样复杂的情况下, 可能要重新考虑是否应该用更适当的方式来表达你的想法。

第二十四章讨论了(象脚注这些)垂直模式的内容从水平列迁移为封装的垂直列的详细规则。插入对象, 标记和 `\vadjust` 的结果都按同样的方式迁移。

现在, 讨论构造象 `\topinsert` 和 `\footnote` 这样的宏的 TEX 原始控制系列。我们将学习 TEX 幕后的低级语言, 使得用户可以对盒子和粘连做更复杂的处理。我们的讨论分两部分: 首先我们讨论 TEX 的"寄存器", 用户可以用它进行排版的相关运算; 接着是出现在水平和垂直列中的插入项目。第一个主题的讨论(寄存器)用一个"危险"标识来标记, 因为寄存器在 TEX 的高级应用中是普遍的, 而不管是否与插入对象有关。而第二个主题用两个"危险"标识标记, 因为插入对象是相当深奥的。

TEX 有 256 个寄存器, 叫做 `\count0` 到 `\count255`, 每个都可以包含在 -2147483647 和 $+2147483647$ 之间的整数(包括这两个值); 即, 此值应该小于 2^{31}。 TEX 还有 256 个寄存器叫做 `\dimen0` 到 `\dimen255`, 每个都可以包含一个 ⟨dimen⟩ (见第十章)。还有另外 256 个寄存器叫做 `\skip0` 到 `\skip255`, 每个都包含 ⟨glue⟩ (见第十二章); 并且还有 `\muskip0` 到 `\muskip255`, 每个都包含 ⟨muglue⟩ (见第十八章)。通过下列方法你可以赋予它们新的值:

`\count`⟨number⟩ `=` ⟨number⟩
`\dimen`⟨number⟩ `=` ⟨dimen⟩
`\skip`⟨number⟩ `=` ⟨glue⟩

---

† Printers often use the symbols `\dag` (†), `\ddag` (‡), `\S` (§), and `\P` (¶) as reference marks; sometimes also `$\|$` (‖). You can say, e.g., '`\footnote\dag{...}`'.

‡ Yet Gibbon's *Decline and Fall* would not have been the same without footnotes.

`\muskip`⟨number⟩ `=` ⟨muglue⟩

而且还可以通过下列方法在其上面加上或减去同样类型的值:

`\advance\count`⟨number⟩ `by` ⟨number⟩
`\advance\dimen`⟨number⟩ `by` ⟨dimen⟩
`\advance\skip`⟨number⟩ `by` ⟨glue⟩
`\advance\muskip`⟨number⟩ `by` ⟨muglue⟩

例如, '`\dimen8=\hsize \advance\dimen8 by 1in`'把寄存器 `\dimen8` 设置为比正常的行的长度多 1 inch。


如果要增加无限大的粘连量, 那么低阶无限大就忽略掉了。例如, 下列两个命令执行后,

```
\skip2 = 0pt plus 2fill minus 3fill
\advance\skip2 by 4pt plus 1fil minus 2filll
```

`\skip2` 的值就是 4 pt plus 2 fill minus 2 filll。


也可以进行乘除, 但是只能用整数。例如, '`\multiply\dimen4 by 3`'就把 `\dimen4` 的值变成 3 倍, '`\divide\skip5 by 2`'就把当前寄存在 `\skip5` 中的 3 个粘连值减半。不能用零除, 也不能使乘法的结果超出寄存器的限制。正整数除以正整数要把余数去掉, 如果有一个改变了正负号, 结果将改变正负号。例如, 14 除以 3 得 4; −14 除以 3 得 −4; −14 除以 −3 得 4。尺寸的值是 sp (scaled points) 的整数倍。


可以在 ⟨number⟩ 的情况下使用任何 `\count` 寄存器, 在 ⟨dimen⟩ 的情况下使用任何 `\dimen` 寄存器, 在 ⟨glue⟩ 下使用 `\skip` 寄存器, 在 ⟨muglue⟩ 使用 `\muskip` 寄存器。例如, '`\hskip\skip1`'把一个粘连放在列中, 其值为 `\skip1` 的值; 并且如果 `\count5` 等于 20, 那么命令'`\advance\dimen20 by\dimen\count5`'就等于'`\multiply\dimen20 by 2`'。


`\dimen` 寄存器还可以用在 ⟨number⟩ 情况下, `\skip` 寄存器可以用在 ⟨dimen⟩ 或 ⟨number⟩。通过忽略掉伸缩量, TEX 把 ⟨glue⟩ 变成 ⟨dimen⟩; 通过设定的 sp (scaled points) 单位, 把 ⟨dimen⟩ 变成 ⟨number⟩。例如, 如果 `\skip1` 的值为 1 pt plus 2 pt, 那么'`\dimen1=\skip1`'将把 `\dimen1` 的值设置为 1 pt; 命令'`\count2=\dimen1`'或'`\count2=\skip1`'将把 `\count2` 的值设置为 65536。这些规则也可用于 TEX 的内部参数; 例如, '`\dimen2=\baselineskip`'将把 `\dimen2` 设置为当前基线间距这个粘连的自然间距量。


▶ **练习15.6**
当下列命令依次执行时, 给出其结果:

```
\count1=50   \dimen2=\count1pt   \divide\count1 by 8
\skip2=-10pt plus\count1fil minus\dimen2
\multiply\skip2 by-\count1  \divide\skip2 by \dimen2  \count6=\skip2
\skip1=.5\dimen2 plus\skip2 minus\count\count1fill
\multiply\skip2 by\skip1   \advance\skip1 by-\skip2
```

▶ **练习15.7**

在下列三个命令执行后, \skip5 是什么?

```
\skip5=0pt plus 1pt
\advance\skip5 by \skip4  \advance\skip5 by -\skip4
```

▶ **练习15.8**

(数学家做)假设 \dimen3 不是零, 怎样得到距 \dimen2 最近的 \dimen3 的倍数?

寄存器要遵守 TEX 的组结构。例如, 在 {...} 中改变 \count3 对外面的 \count3 没有影响。因此, 对每种类型, TEX 的有效寄存器超过 256 个。如果要使寄存器命令延续到组外, 必须在改变其值时使用 \global。

▶ **练习15.9**

在下列一系列命令后 \count1 中的值是多少?

```
\count1=5 {\count1=2 \global\advance\count1by\count1
  \advance\count1by\count1}
```

前十个 \count 寄存器, \count0 到 \count9, 保留为特殊用途: 当输出一个页面时, TEX 将把这十个计数显示在终端上, 并且把它们作为那个页面的识别码传给输出文件。 在终端上这些计数用小数点隔开, 并且去掉后面无用的'.0'。因此, 例如, 如果当页面输出到 dvi 文件时 \count0=5 和 \count2=7, 并且其它计数寄存器为零, TEX 将显示出'[5.0.7]'。Plain TEX 用 \count0 表示页码, 并且保留 \count1 到 \count9 等于零; 这就是第一页输出时所看到的'[1]'。在更复杂的应用中, 页码可以有更多的结构; 十个计数都用上就得到许多识别码。

一般希望把寄存器符号化。 TEX 提供了一个命令 \countdef (类似于 \chardef, 参见第八章), 用它很容易做到: 只需要使用

```
\countdef\chapno=28
```

那么今后 \chapno 就定义为 \count28。对其它类型的数值寄存器, 可以使用类似命令 \dimendef, \skipdef, \muskipdef。在控制系列由 \countdef 定义后, 就可以在 TEX 的命令中使用它了, 就象 \tolerance 这样的整数参数一样。类似地, \dimendef 得到一个新的尺寸参数, \skipdef 得到一个新的粘连参数, \muskipdef 得到一个新的 muglue 参数。

除了数值寄存器, TEX 还有 256 个盒子寄存器, 叫做 \box0 到 \box255。当使用 \setbox⟨number⟩ =⟨box⟩ 时, 盒子寄存器就得到一个值; 例如, '\setbox3=\hbox{A}'把 \box3 设置为包含一个字母 A 的 hbox。\setbox 的几个其它例子已经在第十二章中出现过了。第十章指出, '2\wd3'是一个尺寸, 表示 \box3 的宽度的两倍; 类似地, \ht⟨number⟩ 和 \dp⟨number⟩ 可以用来得到给定盒子寄存器的高度和深度。

就象运算寄存器一样, 盒子寄存器对组也是局部的。但是在盒子寄存器和其它寄存器之间有一个大的差别: 当用到一个盒子时, 它就将其值丢弃。例如, 在水平列中的构造'\raise2pt\box3'不但在把盒子升高 2 pt 后把它放在列中, 而且把 \box3 置空。 TEX 这样做是为了提高效率, 因为要避免重复可能

出现的大盒子的内容。如果要使用盒子寄存器而不擦掉其内容, 只需要使用'\copy'而不是'\box'; 例如, '\raise2pt\copy3'。

使用盒子寄存器的另一种方法是通过使用'\unhbox'来把 hbox 的内容提出来。它象'\box'那样把寄存器的内容擦掉, 并且还去掉了一层盒子。例如, 命令

```
\setbox3=\hbox{A} \setbox3=\hbox{\box3 B}
\setbox4=\hbox{A} \setbox4=\hbox{\unhbox4 B}
```

把 \hbox{\hbox{A}B} 放在 \box3 中, 把 \hbox{AB} 放在 \box4 中。类似地, \unvbox 解开 vbox。如果要递加地构造一个大盒子(比如目录), 最好使用 \setbox4 例子中的 \unhbox 或 \unvbox; 否则就要浪费更多的 TEX 内存空间, 甚至可能得到超出硬件和软件极限的嵌套太深的盒子。

命令 \unhcopy 和 \unvcopy 与 \unhbox 和 \unvbox 的关系类似于 \copy 与 \box。(但是这种命名的确少见。)

去掉盒子的命令同时把盒子外层的任何粘连也去掉了。例如, 看看下面一系列命令:

```
\setbox5=\hbox{A \hbox{B C}} \setbox6=\hbox to 1.05\wd5{\unhcopy5}
```

它使得 \box6 的宽度比 \box5 大 5 个百分点; 在 A 和 \hbox{B C} 之间的粘连伸长来填满, 但是里面盒子的内部粘连不变。

盒子寄存器或者是"置空的", 或者包含一个 hbox 或 vbox。在置空寄存器和包含一个高度宽度深度都是零的空盒子的寄存器之间有区别; 例如, 如果 \box3 是置空的, 那么可以使用 \unhbox3, 或者 \unvbox3, 或者 \unhcopy3, 或者 \unvcopy3, 但是如果 \box3 等于 \hbox{}, 那么只能使用 \unhbox3 或者 \unhcopy3。如果使用'\global\setbox3=⟨box⟩', 那么当后面要用到或去掉盒子时, 寄存器 \box3 将变成"全局置空的"。

▶ **练习15.10**
在下列命令执行后, 寄存器 \box5 里是什么?

```
\setbox5=\hbox{A}  \setbox5=\hbox{\copy5\unhbox5\box5\unhcopy5}
```

▶ **练习15.11**
盒子寄存器编组后, \box3 中是什么? '{\global\setbox3=\hbox{A}\setbox3=\hbox{}}'?

如果对 TEX 怎样操作其寄存器不确定, 可以使用特定的'\show'命令进行实时实验。例如,

```
\showthe\count1  \showthe\dimen2  \showthe\skip3
```

将显示出 \count1, \dimen2 和 \skip3 的内容; 并且'\showbox4'将显示 \box4 的内容。盒子的内容只出现在 log 文件中, 除非使用了'\tracingonline=1'。Plain TEX 提供了一个宏'\tracingall', 它开启了每个可能的相互作用的模式, 包括 \tracingonline。作者利用这些特性可以验证上面几个练习的答案。

TEX 的大量应用采用了不同人编写的不同的宏集。如果寄存器——比如说 \count100——同时在几个宏中为不同目的而使用, 那么就会混乱不堪。因此, plain TEX 提供了一个分配工具; 如果每个宏的编写者使用这些约定, 那么冲突就会变成协作。想法是, 比如当要为一个特殊目的而使用 \count 寄存器时, 使用'\newcount'。例如, 作者设计了一个叫'\exercise'的宏以格式化本手册的练习, \exercise 的一个特性是要计算出当前练习的编号。在附录 E 中的格式化宏为此目的而预约了一个 \count 寄存器, 如下:

```
\newcount\exno
```

并且接下来命令'\exno=0'用在每章的开头。类似地, 每当出现一个新的练习时, 就使用'\advance\exno by1', 并且'\the\exno'用在排版当前练习的编号上。\newcount 命令把一个唯一的计数寄存器指定给其变量 \exno, 并且用 \countdef 命令来定义 \exno。所有的其它格式化宏不知道哪个 \count 寄存器实际上对应于 \exno。

除了 \newcount 之外, plain TEX 还提供了 \newdimen, \newskip, \newmuskip 和 \newbox; 还有 \newtoks, \newread, \newwrite, \newfam 和 \newinsert, 这些我们还没有讨论。附录 B 和 E 给出了正确使用分配的几个例子。在 \newbox, \newread 等情况下, 被分配的数由 \chardef 定义。例如, 如果命令'\newbox\abstract'用来定义一个包含摘要的盒子寄存器, 并且 \newbox 要把 \box45 分配给它来使用, 那么通过'\chardef\abstract=45'来定义 \abstract 的含义。 TEX 允许把 \chardef 的量当整数使用, 这样可以使用 \box\abstract 和 \copy\abstract 等等。(没有 \boxdef 这个命令。)

▶ **练习15.12**

设计一个宏 \note, 它依次得到脚注的编号。例如, [1] 如果输入下列内容, 它将得到这里的脚注[2]:

```
... example,\note{First note.} it should produce
the footnotes here\note{Second note.} if ...
```

(使用 \newcount 为脚注分配一个 \count 寄存器。)

但是有时候, 要用寄存器临时储存一下, 并且确知不会与其它宏冲突。习惯上把寄存器 \count255, \dimen255, \skip255 和 \muskip255 为此保留下来。还有, plain TEX 把 \dimen0 到 \dimen9, \skip0 到 \skip9, \muskip0 到 \muskip9 和 \box0 到 \box9 保留下来随时使用; 这些寄存器从来都不被 \new... 命令分配。我们已经知道, \count0 到 \count9 是特殊的, 并且 \box255 也是特殊的; 所以这些寄存器应该避免使用, 除非知道自己在做什么。

当然, 任何寄存器都可以在组中短期使用(包括 \count0 到 \count9 和 \box255, 还有为其它目的而分配的寄存器), 因为寄存器的变化被限制在组内了。但是, 必须确保在组结束之前 TEX 不输出任何页面, 因为否则输出例行程序会在不合适的时间被调用。只要 TEX 试图从备选内容向当前页面移送东西, 就很有可能调用输出例行程序, 因为接下来它可能发现一个 $c=\infty$ 的分页。下面是可能出现的时间列表: (a). 在段落开头或结尾, 假如此段落被放在主垂直列了。(b). 在这样的段落中的列表公式开头或结

[1] First note.
[2] Second note.

尾。(c). 在垂直模式下完成一个 `\halign` 后。(d). 把盒子或 penalty 或插入对象放在主垂直列后。(e). 在 `\output` 程序结束后。

现在, 借助于 TEX 的灵活的寄存器, 可以详细讨论插入对象了。有 255 类插入对象, `\insert0` 到 `\insert254`, 并且它们与同编号的其它寄存器绑在一起。例如, `\insert100` 与 `\count100`, `\dimen100`, `\skip100` 和 `\box100` 有关。因此, plain TEX 提供了一个插入对象的分配函数, 就象对寄存器一样; 附录 B 包括了命令:

```
\newinsert\footins
```

它把 `\footins` 定义为脚注插入对象的编号。其它处理脚注的命令指有 `\count\footins`, `\dimen\footins`, 等等。浮动的顶部插入的宏类似地是'`\newinsert\topins`', 它把 `\topins` 定义为其分类的编号。每一类插入对象都是独立的, 但是 TEX 一类中保持插入对象的次序。这里, `\footins` 是类 254, `\topins` 是类 253, 但是这些宏不直接使用这样的编号。

为此, 我们讨论一种特殊类型的插入对象, 叫做类 n; 接着我们将处理 TEX 的原始命令

`\insert` n{⟨vertical mode material⟩}

它把插入项目放在水平或垂直列中。对此类插入对象,

`\box` n 是页面输出时要显示的内容;
`\count` n 是分页的放大因子;
`\dimen` n 是每页的最大插入尺寸;
`\skip` n 是在页面上分配的额外空白。

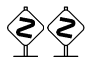
例如, 用 `\insert100` 插入的内容最后出现在 `\box100` 中。

设 `\insert` n 的自然高度加深度为 x; 那么 `\count` n 是 1000 乘以因子, x 通过它对页面排版起作用。例如, plain TEX 设置 `\count\footins=1000`, 因为有一个一对一的关系: 10-point 的脚注会使页面缩短 10 pt。但是如果要让脚注出现在双栏中, 那么计数值为 500 才合适。在附录 E 中, 有一个插入类, 添加边注来进行校对; 在此情形下, 计数值为零。实际上没有进行放大; 在计算各种分页的成本时, `\count` n 只是作笔记的一个数。

页面上第一个脚注要求额外的间距, 因为我们要把脚注与正文分开, 并且要生成一个水平标尺。Plain TEX 设置'`\skip\footins=\bigskipamount`'; 这意味着, 对至少包含一个脚注插入类 `\footins` 的任意页面, 输出例行程序将添加一个 bigskip 的额外间距。

有时候希望给插入的大小一个最大限制; 例如, 大家都不希望整个页面都是脚注。Plain TEX 设置 `\dimen\footins=8in`; 它表示对任何页面, `\box\footins` 不能设定得超过 8 inch 的脚注。

在继续学习之前,可能要回顾一下在本章开头讨论的分页算法。另一方面, 可能你的确不想再阅读本章剩下的内容了。


现在下面是当一个 \insert n 从“备选内容”移到“当前页面”时 TeX 所实行的算法。(记住, 这样的移动并不意味着实际上已经插入了; 当前页面随后将继续增加, 直到最低成本的断点, 并且只有在断点前面的插入对象才实际被执行。) 设 g 和 t 是当前 \pagegoal 和 \pagetotal; 设 q 是当前页面要积累的 \insertpenalties; 设 d 和 z 为当前的 \pagedepth 和 \pageshrink。(d 的值最大为 \maxdepth; 这个值仍未与 t 合并。) 最后, 设 x 是要移到当前页面的 \insert n 的自然高度和深度; 设 f 是相应的放大因子, 即 \count n 除以 1000。

Step 1. 如果当前页面上尚没有 \insert n, 那么 g 将减小 $hf+w$, 其中 h 是 \box n 的当前高度加深度, w 是 \skip n 的自然间距的量; 还有要把 \skip n 的伸缩量包括到当前页面的整体中(特别是, 它会影响 z)。

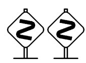
Step 2. 如果当前页面上前面的 \insert n 已经被裂分了, 就把参数 \floatingpenalty 加到 q 上, 并且跳过第 3 和 4 步。

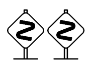
Step 3. 检验一下是否当前插入对象可以放在页面上而毋需裂分。这表示, 当把 \box n 与当前页面上前面所有的 \insert n 加在一起时, \box n 的高度和深度不会超过 \dimen n; 还有, 它意味着 $xf \le 0$ 或者 $t+d+xf-z \le g$。如果这两个测试都通过了, 就从 g 中减去 xf, 并且跳过第 4 步。

Step 4. (当前插入对象要裂分, 至少暂时是这样; 但是如果最低成本页面出现得比目前的插入对象早, 裂分实际上没有发生。) 首先计算最大量 v 使得 v 的高度加深度不能把总插入对象放在比 \dimen n 还大的 \box n 中, 并且使得 $t+d+vf \le g$。 (注意, 在后面的公式中 z 将被忽略, 但是当试着避免裂分时, 在第 3 步将用到可用的伸缩性。) 接着, 找到最低成本的方法把插入对象的垂直列开头裂分, 移得到高度为 v 的盒子。(利用的是类似于分页的算法, 但是没有插入这个复杂的问题; 一个额外的‘\penalty-10000’项目设定出现在垂直列结尾, 以确保存在合理的断点。) 设 u 是最低成本的盒子的高度加深度, r 是对应于最佳断点的 penalty。把 g 减小 uf, 把 q 增加 r。(如果 \tracingpages=1, 那么 log 文件就给出了神秘的信息: ‘% split n to v,u p=r’。例如,

```
% split254 to 180.2,175.3 p=100
```

表示算法试着把 \insert254 裂分为 180.2 pt; 最佳裂分实际上是高度为 175.3 pt, 并且断点的 penalty 是 100。)


这个算法当然是复杂的, 但是好像没有这样更简单的了。注意, 在第 4 步中, 插入对象中的 -10000 的 penalty 很可能导致裂分, 因此在难以处理时, 用户可以给出一些断点的提示。算法提供了各种不同的性质: 通过在每个浮动插入前加入小 penalty 并且把相应的 \floatingpenalty 设为零, 浮动插入对象可以变成裂分插入对象的一种特殊情形; 通过更大的 penalty 和裂分插入(见附录 B), 象脚注这样的非浮动插入对象也可容纳下。


在第 4 步中提到的裂分操作也可以作为一个原始命令使用: ‘\vsplit⟨number⟩ to⟨dimen⟩’通过从盒子寄存器中裂分出给定量的内容而断点一个 vbox。例如,

```
\setbox200=\vsplit100 to 50pt
```

设置 `\box200` 为高度为 50 pt 的 vbox; 它在 `\box100` (它应该是应该 vbox)中的整个垂直列中找到目标高度为 50 pt 的最低成本断点, 其中的 badness 和 penalty 与分页一样(只是 $q = 0$)。此算法使用 `\splitmaxdepth` 来代替 `\maxdepth` 来控制盒子的最大深度。因此, 通过去掉直到最佳断点的所有内容, 并且把紧跟最佳断点的可弃项目包括进来, 它就把 `\box100` 的顶部剪去了; 在 `\box100` 中第一个盒子前, 加上新粘连 `\splittopskip`, 就象出现在页面顶部的粘连 `\topskip` 一样。但是, 如果最佳断点出现在 `\box100` 中垂直列的结尾——'`\penalty-10000`'项目假定出现在那里——或者如果最佳断点后的所有项目都是可弃的, 那么 `\vsplit` 之后 `\box100` 将是置空的。并且如果 `\box100` 在 `\vsplit` 之前是置空的, 那么 `\box100` 和 `\box200` 此后也是置空的。

当 TeX 正在向当前页面输送插入对象时, 最好不要改变 `\box` n, `\count` n, `\dimen` n, 或 `\skip` n, 因为 TeX 的算法假定这些量是静态的。但是可以改变 `\floatingpenalty`, `\splittopskip` 和 `\splitmaxdepth`; 当 TeX 裂分或浮动插入对象时, 要用到这些值, 它们只是在'`\insert` n`{...}`'的右括号内才是当前值。例如, 附录 B 在脚注插入中使用 `\floatingpenalty=20000`, 以防止脚注在其它插入开始前裂分, 但是在浮动顶部插入时 `\floatingpenalty=0`。附录 B 还用到 `\splittopskip` 和 `\splitmaxdepth` 的特殊值, 以及支架, 这样裂分的脚注与未裂分脚注的排版间距就一致了。

宏 `\footnote` 在独立的水平列中放了一个 `\insert`。在段落断行后, 这个插入对象就移出来放到包含它的行后面的垂直列中(见第十四章)。因为在那个盒子(即那个行)和插入对象之间没有合理的断点, 所以 TeX 将把插入对象放在其所对应行所在的页面上。

▶ **练习15.13**
仔细研究一下分页算法。脚注可能与其参照点不出现在同一页面吗?

当最后选定最佳分页后, TeX 从"当前页面"底部把选定断点后的所有东西都去掉, 送回到"备选内容"的顶部。选定的断点本身被放在备选内容的正顶部。如果它是一个 penalty 项目, 那么 penalty 的值就记录在 `\outputpenalty` 中, 并且备选列中的 penalty 被变为 10000; 否则, `\outputpenalty` 被设置为 10000。当前页面剩下的插入对象有三种: 对每个类 n, 有未裂分的插入对象, 接着可能是一个裂分的插入对象, 再接着是其它的。如果 `\holdinginserts` > 0, 那么每个插入对象都保持在适当的位置(使得它们可能被再次输送); 否则它们将如下从当前页面列中全部去掉: 未裂分的插入对象追加到 `\box` n, 而且它们之间没有行间粘连。(要用到支架, 就象在附录 B 中的宏 `\vfootnote` 一样。) 如果是裂分的插入对象, 那么它 `\vsplit` 为前面第 4 步计算的大小; 上部分被当做未裂分的处理, 剩下的(如果还有的话)转换为插入对象, 就象没有裂分过一样。跟在任意其它同类浮动插入对象后面的这个剩下的对象, 继续保持在单独的地方。(如果在 `\output` 例行程序正在执行时使用 `\showlists`, 它们将出现在"当前页面"上; 在 `\output` 例行程序期间, 这样的插入对象的总数出现在 `\insertpenalties` 中。) 最后, 在当前页面断点之前的剩余项目被一起否则高度为 g 的 `\vbox` 中, 其中 g 是那时断点的 `\pagegoal`, 用到了所保存的 `\maxdepth` 的值; 这个盒子变成了 `\box255`。现在, 用户的 `\output` 例行程序进入了 TeX 扫描(见第二十三章); 它的任务是基于 `\box255` 和任意它得到的插入对象盒子的内容来组成最后的页面。输出例行程序可能把这些盒子解开, 使得它们的粘连重新设定; 当凑巧时, 在插入对象盒子中的粘连一般与页面其它的粘连能很好地合并。在 `\output` 例行程序结束后, 保持的插入项目被首先放在备选内容列上, 接着是由 `\output` 构建的垂直列, 接着是以分页开始的备选内容。(深呼吸一下。) 你看懂了吗?

16. 数学公式

设计 TEX 就是为了能够用简单的输入方法来处理复杂的数学公式。基本想法是, 一个复杂的公式按照一个简单的方法由不太复杂的公式组成; 而不太复杂的公式又是由较简单的公式组成; 等等。从另一个角度来看, 如果知道怎样输入简单公式和怎样把它们组合成大公式, 那么就能灵活地处理任何公式了。因此, 我们从简单公式开始, 逐步展开我们的方法。

最简单的公式就是象'x'这样的单个字母, 或者象'2'这样的单个数字。为了把它们输入到 TEX 文本中, 应该分别键入'`$x$`'和'`$2$`'。注意, 所有的数学公式都要用特殊的数学括号封装起来; 在本手册中, 按照附录 B 中定义的 plain TEX 格式, 把 `$` 作为数学括号使用, 因为料想数学内容很多。

当键入'`$x$`'时, 'x'以 italic 格式出现, 但是当键入'`$2$`'时, '2'以 roman 格式出现。一般地, 按照数学排版的通常惯例, 键盘上的所有字符在数学公式中都有专门的解释: 在这里, 字母是 italic 字母, 而数字和标点是 roman 数字和标点; 连字符(`-`)是负号($-$), 它与破折号几乎一样但是有点差别(见第二章)。键入第一个 `$` 就进入了"数学模式", 键入第二个就跳出数学模式了(见第十三章)。因此, 如果忘记了一个 `$` 或多输入一个 `$`, 那么 TEX 可能因不知所措而错误百出。

在数学家看来, 不熟悉数学的人排版出来的数学公式一般看起来怪怪的, 因为新手一般把间距都弄错了。为了缓和这种矛盾, TEX 在数学公式中设置了自己的间距; 并且它忽略掉你个人在 `$` 之间输入的任何空格。例如, 如果输入的是'`$ x$`'和'`$ 2 $`', 那么它们与'`$x$`'和'`$2$`'是一样的。可以输入'`$(x + y)/(x - y)$`'或者'`$(x+y) / (x-y)$`', 但是它们两个的结果都是'$(x+y)/(x-y)$', 在这个公式中, 在符号 $+$ 和 $-$ 两边有一点额外的间距, 但是在符号 / 两边却没有。因此, 你不需要记住复杂的数学间距规则, 并且可以随心所欲地使用空格。当然, 空格还要按照正常的方法来标记出控制系列的结尾, 就象在第三章中讨论的那样。在大多数情况下, TEX 的间距都符合数学家的习惯; 但是在第十八章将看到, 如果需要的话, 可以用控制系列来改变 TEX 的间距。

数学家喜欢做的一件事就是在公式使用希腊字母。在 plain TEX 格式中, 输入'`$$\alpha, \beta, \gamma, \delta;$$`'就可得到前四个希腊字母

$$\alpha, \beta, \gamma, \delta;$$

还有, 象'Γ'这样的大写希腊字母可以输入'`$\Gamma$`'来得到。如果对希腊字母还不熟悉也别有压力; 如果需要的话很容易就掌握它们了。唯一的困难是某些看起来几乎一样的字符却要细心区分开。例如, 不要把希腊字母 `\nu` (ν) 和 `\kappa` (κ) 同 italic 字母 v 和 x 混淆; 希腊字母 `\phi` (ϕ) 与"空集"符号 `\emptyset` ($\emptyset$) 不同。小写 epsilon (ϵ) 与集合论中的"属于"符号($\in$)不同; '`$\epsilon$`'得到的是 ϵ, '`$\in$`'得到的是 $\in$。在 plain TEX 的数学 italic 字体中, 某些小写希腊字母有变体: '`$(\phi,\theta,\epsilon,\rho)$`'得到'$(\phi,\theta,\epsilon,\rho)$'而'`$(\varphi,\vartheta,\varepsilon,\varrho)$`'得到'$(\varphi,\vartheta,\varepsilon,\varrho)$'。

除了希腊字母外, 还有许多有意思的符号, 象'$\approx$'(其输入为'`$\approx$`')和'$\mapsto$'(输入为'`$\mapsto$`')。这些控制希腊及其对应的字符的完整列表在附录 F 中给出。这样的控制系列只允许在数学模式下使用, 即在 `$` 之间, 因为相应的符号来自数学字体。

▶ **练习16.1**

怎样得到下列公式'$\gamma+\nu\in\Gamma$'?

▶ **练习16.2**

在附录 F 中找出‘$\le$’, ‘$\ge$’和 ‘$\ne$’的控制系列。(它们可能是不在键盘上的最常用的三个数学符号了。) 在 plain TeX 中其名称是什么?

现在我们讨论怎样从简单公式来构建更复杂的公式。首先, 通过使用‘`^`’和‘`_`’, 可以得到上标 $^{\text{(up high)}}$ 和下标 $_{\text{(down low)}}$, 如下例所示:

输入	输出
`$x^2$`	x^2
`$x_2$`	x_2
`$2^x$`	2^x
`$x^2y^2$`	x^2y^2
`$x ^ 2y ^ 2$`	x^2y^2
`$x_2y_2$`	x_2y_2
`$_2F_3$`	${}_2F_3$

注意, `^` 和 `_` 只作用在下一个单个字符上。如果要在上标或下标中放几个字符, 就要把它们封装在大括号中:

`$x^{2y}$`	x^{2y}
`$2^{2^x}$`	2^{2^x}
`$2^{2^{2^x}}$`	$2^{2^{2^x}}$
`$y_{x_2}$`	y_{x_2}
`$y_{x^2}$`	y_{x^2}

在这些例子中的大括号是用来确定“子公式”的, 即大公式的简单组分。 TEX 把每个子公式做成一个盒子, 并且把盒子看作一个单个字符。括号还有通常的编组功能, 就象第五章中讨论的那样。

输入‘`x^y^z`’或‘`x_y_z`’是不合法的; TEX 将为“双上标”或“双下标”而抱怨。为了明确表达出你的意思, 必须输入‘`x^{y^z}`’或‘`x^{yz}`’或‘`x_{y_z}`’或‘`x_{yz}`’。

如果上标或下标后面跟的是一个字符, 就只作用在字符上; 但是当跟的是一个子公式时, 就作用在整个子公式上, 并且它将因此而升高或降低。例如,

`$((x^2)^3)^4$`	$((x^2)^3)^4$
`${({(x^2)}^3)}^4$`	${({(x^2)}^3)}^4$

在第一个公式中, ‘`^3`’和‘`^4`’是右小括号的上标, 即, 是直接在它们前面的字符‘`)`’的上标, 但是在第二个公式中, 它们是封装在大括号中子公式的上标。第一个更好一些, 因为它容易输入和理解。

如果上标或下标前面什么也没有(就象前页的例子‘`_2F_3`’那样, ‘`_2`’前面什么也没有), 那么它就是一个空子公式的上标或下标。这样的情况在数学中很少(幸好); 但是如果真遇到它, 最好用大括号明确给出空子公式来表达你的意思。换句话说, 要在公式中得到‘${}_2F_3$’, 最好输入‘`{}_2F_3`’或‘`{_2}F_3`’或‘`{_2F_3}`’。

▶ 练习16.3

在‘`$x + _2F_3$`’和‘`$x + {}_2F_3$`’的结果之间有没有差别？有的话，是什么？

▶ 练习16.4

给出‘`${x^y}^z$`’和‘`$x^{y^z}$`’之间的差别。

你可以同时给出上下标，并且可以用任意次序：

`$x^2_3$`	x^2_3
`$x_3^2$`	x_3^2
`$x^{31415}_{92}+\pi$`	$x^{31415}_{92}+\pi$
`$x_{y^a_b}^{z_c^d}$`	$x_{y^a_b}^{z_c^d}$

注意，同时使用上下标时上标在下标上方居左。但是，当在某些字母后面时，下标会“挤进去”一点；例如，‘`$P_2^2$`’得到的是‘P_2^2’。如果要使得上下标的左边界对齐，可以通过插入空的子公式来欺骗 TEX：‘`$P{}_2^2$`’得到的就是‘$P{}_2^2$’。

控制系列 `\prime` 得到的是符号‘$\prime$’，它一般用作上标。实际上，‘$\prime$’太大，一般只作为上下标使用，这时它将以较小的尺寸出现。下面是一些典型的例子：

输入	dv
`$y_1^\prime$`	$y_1^\prime$
`$y_2^{\prime\prime}$`	$y_2^{\prime\prime}$
`$y_3^{\prime\prime\prime}$`	$y_3^{\prime\prime\prime}$

因为单个和两个撇号经常要用到，所以 plain TEX 提供了一种方便的简写：可以直接输入 `'` 来代替 `\prime`，用 `''` 代替 `\prime\prime`，等等。

`$f'[g(x)]g'(x)$`	$f'[g(x)]g'(x)$
`$y_1'+y_2''$`	$y_1'+y_2''$
`$y'_1+y''_2$`	$y'_1+y''_2$
`$y'''_3+g'^2$`	$y'''_3+g'^2$

▶ 练习16.5

想想为什么 TEX 把 `\prime` 做成大个字符却只让出现在上标中，而不是把它变成小个字符直接提高到上标的位置上？

▶ 练习16.6

有时候数学家用到的“张量符号”要求上标和下标错开，就象‘$R_i{}^{jk}{}_l$’这样。怎样才能得到这种结果？

简单公式组合成复杂公式的另一种方法是利用控制系列 `\sqrt`, `\underline` 或 `\overline`。象 `^` 和 `_` 一样, 这些命令也作用在其后的字符或子公式上:

`$\sqrt2$`	$\sqrt2$
`$\sqrt{x+2}$`	$\sqrt{x+2}$
`$\underline4$`	$\underline4$
`$\overline{x+y}$`	$\overline{x+y}$
`$\overline x+\overline y$`	$\overline x+\overline y$
`$x^{\underline n}$`	$x^{\underline n}$
`$x^{\overline{m+n}}$`	$x^{\overline{m+n}}$
`$\sqrt{x^3+\sqrt\alpha}$`	$\sqrt{x^3+\sqrt\alpha}$

也可以得到 3 次方根号'$\sqrt[3]{\ }$', 类似的要用到 `\root`:

`$\root 3 \of 2$`	$\sqrt[3]{2}$
`$\root n \of {x^n+y^n}$`	$\sqrt[n]{x^n+y^n}$
`$\root n+1 \of a$`	$\sqrt[n+1]{a}$

命令 `\sqrt`, `\underline` 和 `\overline` 可以把直线放在任意尺寸和形状的子公式上面或下面; 横线可以改变其长度和位置, 使得长的足以盖住后面的子公式, 高或低到足以使子公式碰不到它。例如, 看看'`\overline l`' ($\overline l$)与'`\overline m`' ($\overline m$): 第一个是短横线, 但是比第二个要高。类似地, '`\underline y`' ($\underline y$)中的横线比'`\underline x`' ($\underline x$)要低; 在 $\sqrt a+\sqrt d+\sqrt y$ 中, 由于各个字母高度和深度各异, 所以根号出现在不同位置上。 TEX 知道每个字母和每个子公式的高度, 深度和宽度, 因为正如在第十一章中讨论的那样, 它把这些都看作盒子。如果公式中只有一个 `\sqrt`, 或者只有一个 `\overline` 或 `\underline`, 那么这样的规则就是完美的了; 但是有时候要求在一个复杂公式的不同组分之间位置一致。例如, 在'$\sqrt{\mathstrut a}+\sqrt{\mathstrut d}+\sqrt{\mathstrut y}$', 要求其中根号的垂直位置是相同的。有一个简单的办法来实现它, 就是利用控制系列 `\mathstrut`, 如下:

```
$\sqrt{\mathstrut a}+\sqrt{\mathstrut d}+\sqrt{\mathstrut y}$.
```

`\mathstrut`是一个看不见的盒子, 其宽度为零, 高度和深度是圆括号'('的高度和深度。因此, 如果没有更复杂的象上下标那样的构造, 包含 `\mathstrut` 的子公式总有相同的高度和深度。第十八章讨论了 `\smash` 和 `\phantom` 这些更强大的命令, 通过它可以完全控制根号和类似符号的位置。

▸ **练习16.7**

测验一下对本章到现在的内容你理解的怎么样, 看看怎样输入下列公式。(一定要与附录 A 的答案核对一下。)

$$10^{10} \qquad 2^{n+1} \qquad (n+1)^2 \qquad \sqrt{1-x^2} \qquad \overline{w+\overline z} \qquad p_1^{e_1} \qquad a_{b_{c_{d e}}} \qquad \root 3 \of {h''_n(\alpha x)}$$

▶ **练习16.8**

输入下列内容后会出现什么错误?

```
If$ x = y$, then $x$ is equal to $y.$
```

▶ **练习16.9**

看看怎样输入下列句子:

Deleting an element from an n-tuple leaves an $(n-1)$-tuple.

▶ **练习16.10**

列出所有低于基线的 italic 字母。(对这些字母, `\underline` 要降低其横线。)

我们已经说过, 在数学模式下输入的字符都有专门的意思, 但是迄今为止的举例还不够; 还不能反映输入 `$` 后全部的效果。现在回到基本问题: 我们系统地总结一下当字符用在公式中时的情况。

52 个字母(`A` 到 `Z` 和 `a` 到 `z`)表示 italic 符号(A 到 Z 和 a 到 z), 数学家称其为“变量”, TEX 却只把它称为“普通符号”, 因为它们是数学公式的主体。在 plain TEX 中, 有两个小写 L 的变体, 即‘l’(输入为‘`l`’)和‘ℓ’(输入为‘`\ell`’)。虽然数学家一般在手稿中写作‘ℓ’, 但是这只是为了与数字‘1’区分开。在打印的数学文稿中, 不需要区分它们, 因为 italic ‘l’与‘1’很容易区分; 因此, 一般使用的是‘l’, 除非特别要求使用‘ℓ’。

Plain TEX 还把下列 18 个字符看作普通符号:

```
0 1 2 3 4 5 6 7 8 9 ! ? . | / ‘ @ "
```

即, 当这些符号出现在字母后面或这 18 个字符后面时, 不插入任何额外的间距。但是与字母不同的是, 当出现在公式中时, 这 18 个字符仍然是 roman 字体。对你而言, 毋需特别去记它们, 除了竖线‘|’有我们后面要讨论的特殊的用法外。还有, 应当注意区分字母‘O’和零: italic 字母 O 几乎从不出现在公式中, 除了出现在左圆括号的前面, 比如‘$O(n)$’; 而数字 0 从不出现在左圆括号的前面, 除非它前面是一个数字, 比如‘$10(n-1)$’。看看左圆括号就 $0K$ 了。(小写 o 好像也只出现在左圆括号前面; 输入的是‘`x_0`’而不是‘`x_o`’, 因为公式‘x_0’一般比‘x_o’更正确。

`+`, `-` 和 `*` 这三个字符称为“二元运算”, 因为它们作用在公式的两个组分上。例如, `+` 是一个加号, 用于求两个数的和; `-` 是减号。星号(`*`)很少用在数学中, 但是它也是一个二元运算。下面的例子表明了当这些二元符号出现在普通符号后面时 TEX 是怎样排版它们的:

输入	输出
`$x+y-z$`	$x+y-z$
`$x+y*z$`	$x+y*z$
`$x*y/z$`	$x*y/z$

注意, `-` 和 `*` 所得到的数学符号与普通文本中所得到的很不一样: 连字符(-)变成减号($-$), 而高高的星号(*)降下来变成了低一些的($*$)。

TEX 不把 / 看作二元运算, 虽然斜线表示的是除法(它在数学领域的确是一个二元运算)。原因是排版者一般在符号 +, − 和 ∗ 两边添加额外间距, 但是在 / 两边不添加。如果 TEX 把 / 按照二元运算进行排版, 那么公式'`$1/2$`'得到的是'1 / 2', 这并不好; 因此 TEX 把 / 看作普通符号。

附录 F 列出了更多的二元运算, 输入它们需要用控制系列而不是单个字符。下面是一些例子:

`$x\times y\cdot z$`	$x\times y\cdot z$
`$x\circ y\bullet z$`	$x\circ y\bullet z$
`$x\cup y\cap z$`	$x\cup y\cap z$
`$x\sqcup y\sqcap z$`	$x\sqcup y\sqcap z$
`$x\vee y\wedge z$`	$x\vee y\wedge z$
`$x\pm y\mp z$`	$x\pm y\mp z$

重要的是把 × (`\times`) 与 X (`X`) 和 x (`x`); 把 ∪ (`\cup`) 与 U (`U`) 和 u (`u`); 把 ∨ (`\vee`) 与 V (`V`) 和 v (`v`); 把 ∘ (`\circ`) 与 O (`O`) 和 o (`o`) 区分开。符号'∨'和'∧'可以叫做 `\lor` 和 `\land`, 因为它们常常表示称为"逻辑或(logical or)"和"逻辑与(logical and)"的二元运算。

顺便说一下, 如果二元运算不出现在它们要操作的两个量之间, 那么就看作普通符号。例如, 在下列情形下, 在 +, − 和 ∗ 两边没有添加额外的间距:

`$x=+1$`	$x=+1$
`$3.142-$`	$3.142-$
`$(D*)$`	$(D*)$

再考虑下列例子, 它们表明, 在上下标中, 二元运算被看作普通符号:

`$K_n^+,K_n^-$`	K_n^+,K_n^-
`$z^*_{ij}$`	z^*_{ij}
`$g^\circ \mapsto g^\bullet$`	$g^\circ \mapsto g^\bullet$
`$f^*(x) \cap f_*(y)$`	$f^*(x) \cap f_*(y)$

▶ **练习16.11**
怎样得到公式'z^{*2}'和'$h_*'(z)$'?

Plain TEX 把 =, <, > 和 : 这四个字符看作"关系符号", 因为它们表达了两个量之间的关系。例如, '$x<y$'的意思是 x 小于 y。这样的关系与象 + 那样的二元运算的意思差别很大, 并且符号在排版时也有些不同:

`$x=y>z$`	$x=y>z$
`$x:=y$`	$x:=y$
`$x\le y\ne z$`	$x\le y\ne z$

```
$x\sim y\simeq z$
```
$x \sim y \simeq z$

```
$x\equiv y\not\equiv z$
```
$x \equiv y \not\equiv z$

```
$x\subset y\subseteq z$
```
$x \subset y \subseteq z$

(后几个例子中出现了很多其它关系符号, plain TEX 通过控制系列来调用它们; 见附录 F。)

逗号‘,’和分号‘;’这两个字符被看作公式中的标点; 这意味着 TEX 在它们后面添加额外的小间距, 但是在前面没有。

```
$f(x,y;z)$
```
$f(x,y;z)$

习惯上在数学公式的句点‘.’后面不添加额外间距, 因此 TEX 把句号看作一个普通符号。如果要把字符‘:’看作标点符号而不是关系符号, 就要用 \colon:

```
$f:A\to B$
```
$f : A \to B$

```
$f\colon A\to B$
```
$f\colon A \to B$

如果要把逗号看作普通符号(即当出现在科学计数法中时), 就把它放在大括号中; TEX 把大括号中的任何东西都看作普通符号。比如,

```
$12,345x$
```
$12, 345x$ (错)

```
$12{,}345x$
```
$12{,}345x$ (对)

▶ **练习16.12**

怎样用简单的方法得到一个小数, 其小数点是升高了的(比如‘3·1416’)?

现在我们已经讨论了字母, 其它普通符号, 二元运算, 关系符号和标点符号; 因此我们几乎覆盖了打字机上的每个键。只是还有几个: 字符‘(’和‘[’称为“开符号”, 而‘)’和‘]’称为“闭符号”; 它们非常象普通符号, 但是当二元运算不真是用在二元关系上时, 它们辅助 TEX 来判断出来。接着还有一个字符 ', 我们知道它是 \prime 上标的缩写。最后我们知道 plain TEX 保留下其它十个字符:

```
\ $ % # & ~ { } _ ^
```

如果它们的 \catcode 值不改变(见第七章), 那么它们在数学模式下不便于当作符号使用。虽然 { 和 } 用于编组, 但是控制系列‘\{’和‘\}’可以用来得到作为开符号的‘{’和作为闭符号的‘}’。

所有这些数学模式的含义都容易改变, 因为每个字符都有一个 \mathcode, 就象在第十七章中讨论的那样; TEX 没有建立不可改变的约定。但是它们的大多数是如此标准以致于改变它们是不明智的, 当然改变 ‘, " 和 @ 的含义除外。

表示上下标的特殊字符 ^ 和 _ 不能在公式外面使用。类似地, 象 \alpha 和 \approx 这样的数学符号的名字, 和象 \overline 这样的数学运算的控制系列也不能进入普通文本。在输入中落掉 $ 符号时, TEX 在错误百出之前就利用这些规则来检测出这个失误。例如, 假设输入的是

```
The smallest $n such that $2^n>1000$ is~10.
```

TEX 不知道在第一个'n'落掉了一个'$', 因为它不懂英语; 因此它在前两个 $ 符号之间发现一个数学公式:

The smallest $nsuchthat$

之后它把'2'看作文本的一部分。但是接着 ^ 引出了矛盾; TEX 将自动在 ^ 前面插入一个 $, 并且产生一个错误信息。这样计算机就回到正常状态了, 并且就象什么也没有发生一样继续处理剩下的内容。

反过来, 空行或 \par 也不能出现在数学模式下。这样 TEX 就可以用另一种方法来修复落掉 $ 这个失误了; 这个错误将被限制在它所出现的段落中。

如果由于某些原因不能用 ^ 和 _ 来表示上下标, 因为可能使用的键盘不通用, 或者因为需要把 ^ 保留给法语重音或其它用处, 那么 plain TEX 提供了替代方法 \sp 和 \sb。例如, '$x\sp2$'是得到'x^2'的另一种方法。另一方面, 有些人很幸运, 键盘上的字符比标准 ASCII 还多。当可以使用这些符号时, 输入 TEX 数学公式更轻松。例如, 作者的键盘上有能产生可见符号的 ↑ 和 ↓(它们使屏幕上在上下标更漂亮); 还有关系符号≤, ≥ 和 ≠(它们可以节省时间); 以及两打偶尔要用到的键。

数学家偏爱在字母上面加重音, 因为这常常是表明数学对象之间关系的有效方法, 再者因为它在不增加所需字体数量的情况下可以大大扩展可以符号的数量。第九章讨论了在普通文本上使用重音, 但是数学重音有些不一样, 因为间距是不同的; 对公式中的重音, TEX 使用特殊的约定, 使得两种重音互相不冲突。下面的数学重音由 plain TEX 提供:

`$\hat a$`	$\hat a$
`$\check a$`	$\check a$
`$\tilde a$`	$\tilde a$
`$\acute a$`	$\acute a$
`$\grave a$`	$\grave a$
`$\dot a$`	$\dot a$
`$\ddot a$`	$\ddot a$
`$\breve a$`	$\breve a$
`$\bar a$`	$\bar a$
`$\vec a$`	$\vec a$

对于前九个, 出现在文本中时分别称为 \^, \v, \~, \', \`, \., \", \u 和 \=; \vec 是只出现在公式中的重音。如果在公式中使用 \^ 或 \v 等, 或者在普通文本中使用 \hat 或 \check 等, TEX 就会提醒。

一般把常用的带重音的字母定义为专门的控制系列是明智之举。例如, 可以把

```
\def\Ahat{{\hat A}}
\def\chat{{\hat c}}
\def\scheck{{\check s}}
\def\xtilde{{\tilde x}}
\def\zbar{{\bar z}}
```

放在文稿开头, 如果每个符号 $\hat{A}$, $\hat{c}$, $\check{s}$, $\tilde{x}$ 和 $\bar{z}$ 用到超过(比如说) 5 次, 那么就节省了很多劳动量, 并且文稿容易阅读。第二十章讨论了怎样定义控制系列。

当在数学公式中在字母 i 和 j 上加重音时, 重音下面要用到无点的字母 $\imath$ 和 $\jmath$。在 plain TEX 中这些符号叫做 `\imath` 和 `\jmath`。因此, 比如论文中要用到'$\hat\imath$'和'$\hat\jmath$', 就应当如下定义:

```
\def\ihat{{\hat\imath}}
\def\jhat{{\hat\jmath}}
```

可以在重音上面放重音, 所得到的符号象 $\hat{\hat{A}}$ 这样, 这可能使得数学家高兴得直叫。但是, 要把上面的重音放在最佳位置需要一点技巧, 因为数学字体的设计者一般告诉 TEX 按照专门的方法把数学重音放在特定的字母上。Plain TEX 提供了一个叫做 `\skew` 的控制系列, 它使得把上面的重音移到正确的位置变得相当简单。例如, '`\skew6\hat\Ahat`'就是用来得到上面的符号。本例中的数字'`6`'是反复试验出来的; '`5`'让上面的重音太靠左了, 而'`7`'又太靠右了, 至少作者看来是如此。想法就是调节 skew 的量直到你满意为止。

实际上, 可以把数学重音放在任何子公式上, 不仅仅是单个字符或重音字符上。但是这样做一般不太好, 因为 TEX 仅仅就是把重音放在整个子公式上方正中。例如, '`$\hat{I+M}$`'得到的是'$\hat{I+M}$'。特别是, 重音 `\bar` 的长度变成不变; 它不象 `\overline` 那样随其下公式的增长而增长。即使应用于单个字符, 有些人也希望用 `\overline` 得到较长的横线; 例如, '`$\bar z+\overline z$`'得到的是'$\bar z+\overline z$', 在定义 `\zbar` 时你可以择优使用。 但是 plain TEX 的确给出了两个可伸长的重音; 它们叫做 `\widehat` 和 `\widetilde`:

`$\widehat x,\widetilde x$`	$\widehat x,\widetilde x$
`$\widehat{xy},\widetilde{xy}$`	$\widehat{xy},\widetilde{xy}$
`$\widehat{xyz},\widetilde{xyz}$`	$\widehat{xyz},\widetilde{xyz}$

这里的第三个例子显示的是可用的最大尺寸。

▶ **练习16.13**

本章也很长; 但是你学会很多! 为了证明, 解释一下怎样得到下列公式: e^{-x^2}, $D\sim p^\alpha M+l$ 和 $\hat g\in(H^{\pi_1^{-1}})'$。(在这个最后的例子中, 假设控制系列 `\ghat` 已经被定义, 使得 `\ghat` 得到的是重音字母 $\hat g$。)

17. 数学排版进阶

数学家爱做的另一件事就是构造分数——并且喜欢把用各种各样的方法把符号放在其它符号上面:

$$\frac{1}{2} \quad 和 \quad \frac{n+1}{3} \quad 和 \quad \binom{n+1}{3} \quad 和 \quad \sum_{n=1}^{3} Z_n^2.$$

通过输入'`$$1\over2$$`'和'`$$n+1\over3$$`'和'`$$n+1\choose3$$`'和'`$$\sum_{n=1}^3 Z_n^2$$`'可以把它们变成列表方程; 本章我们将讨论这些构造的简单规则。

首先我们讨论分数, 它用到命令'`\over`'。控制系列 `\over` 要作用到公式中的所有内容, 除非你把它用大括号封装在一个规定的子公式中; 在后一种情况下, `\over` 只作用于那个子公式的所有内容。

输入	输出
`$$x+y^2\over k+1$$`	$\frac{x+y^2}{k+1}$
`$${x+y^2\over k}+1$$`	$\frac{x+y^2}{k}+1$
`$$x+{y^2\over k}+1$$`	$x+\frac{y^2}{k}+1$
`$$x+{y^2\over k+1}$$`	$x+\frac{y^2}{k+1}$
`$$x+y^{2\over k+1}$$`	$x+y^{\frac{2}{k+1}}$

不允许在同一子公式中使用两次 `\over`; 不能输入象'`a \over b \over 2`'这样的内容, 必须给出 over 作用的范围:

`$${a\over b}\over 2$$`	$\frac{\frac{a}{b}}{2}$
`$$a\over{b\over 2}$$`	$\frac{a}{\frac{b}{2}}$

不幸的是, 这两种方法看起来都很别扭。在数学家开始用 TeX 排版时, 总爱"过度使用" `over`。只要当所构建的东西太小或太拥挤时, 好的排版者或编辑就把分数变成"除式"。例如, 最后两种情况可以写作:

`$$a/b \over 2$$`	$\frac{a/b}{2}$
`$$a \over b/2$$`	$\frac{a}{b/2}$

转换到除式需要一点数学常识, 因为为了不改变公式的意思, 有时候要插入圆括号。除了用'/'代替'`\over`'外, 分数的分子和分母应该放在括号中, 除非它们是单个字符; 例如, $\frac{a}{b}$ 就直接变成 a/b, 但是 $\frac{a+1}{b}$ 要变成 $(a+1)/b$, $\frac{a+1}{b+1}$ 要变成 $(a+1)/(b+1)$。还有, 如果分数的前面有东西, 那么整个分数应放在括号中; 例如,

$\frac{a}{b}x$ 变成 $(a/b)x$。作为没有数学常识的排版者, 在不清楚时应该询问作者; 也可以巧妙地建议在以后的文稿中尽量不出现不好看的分数。

▶ **练习17.1**
怎样更好地排版出公式$x+y^{\frac{2}{k+1}}$?

▶ **练习17.2**
把'$\frac{a+1}{b+1}x$'转换为除式。

▶ **练习17.3**
当输入'`$$x = (y^2\over k+1)$$`'时会出现什么问题?

▶ **练习17.4**
怎样得到'$7\frac{1}{2}$¢'? (假定控制系列 `\cents` 得到的就是'¢'。)

上面的例子表明, 当字母和其它符号出现在分数中时, 有时候会变得更小, 就象把它们放在指数上那么小。现在我们应该讨论一下 TEX 怎样来选择符号的大小。当处理公式时, TEX 实际上有八种不同的字体, 即,

列表字体	(用在行中单独的列表公式中)
文本字体	(用在嵌入文本的公式中)
标号字体	(用于公式的上下标)
小标号字体	(用于公式的二阶上下标)

以及四种其它的"近似"字体, 它们与上面四种几乎一样, 只是指数升高得不那么多。为了简化, 我们用

D, D', T, T', S, S', SS, SS',

来表示这八种字体, 其中 D 是列表字体, D' 是近似的列表字体, T 是文本字体等等。 TEX 还使用数学字体的三种不同大小, 分别叫做文本尺寸, 标号尺寸, 小标号尺寸。

用 TEX 排版公式的正常方法是把它封装在符号 `$...$` 中; 这样得到的是文本字体(字体 T)。或者封装在符号 `$$...$$` 中, 这样得到的是列表字体(字体 D)。当然公式的子公式使用的可能是不同的字体。一旦知道了字体, 就可以确定 TEX 要用的字体的大小:

如果字母的字体是	那么设定的大小为	
D, D', T, T'	文本尺寸	(like this)
S, S'	标号尺寸	(like this)
SS, SS'	小标号尺寸	(like this)

没有"SSS"字体或者"小小标号"字体; 这样小的符号比小标号字体更难看清。因此 TEX 把小标号尺寸作为最小的:

公式的字体	上标的字体	下标的字体
D, T	S	S'

D', T'	S'	S'
S, SS	SS	SS'
S', SS'	SS'	SS'

例如, 如果 `x^{a_b}` 用字体 D 排版, 那么 `a_b` 就用字体 S, `b` 就用字体 SS'; 结果为: ‘x^{a_b}’.

现在我们还没发现字体 D 和字体 T 之间的区别。实际上, 虽然在两种情况下指数都用标号尺寸, 但是其位置有点不同: 看看字体 D 中的 x^2 和 T 中的 x^2 以及 D' 或 T' 中的 x^2 就明白了。但是当处理分数时, 字体 D 和字体 T 之间有一个明显的差别:

公式 α`\over`β 的字体	分子的字体	分母的字体
D	T	T'
D'	T'	T'
T	S	S'
T'	S'	S'
S, SS	SS	SS'
S', SS'	SS'	SS'

因此, 如果在文本中输入‘`$1\over2$`’那么得到的是 $\frac{1}{2}$, 即字体 S' 上是字体 S; 但是如果输入 ‘`$$1\over2$$`’ 就得到列表公式

$$\frac{1}{2}$$

其中字体 T' 上是字体 T。


在这里我们最好还是给出全部的字体规则: `\underline` 不改变字体。数学重音和 `\sqrt` 运算与 `\overline` 把非近似的字体变成相应的近似字体; 例如, D 变成 D', 但是 D' 保持不变。


▶ **练习17.5**
假定公式 $\sqrt{p_2^{e'}}$ 是字体 D, 给出其每一部分的字体和大小。

如果不喜欢 TEX 自动选择的字体, 那么可规定你所要的字体, 只要输入 `\displaystyle`, `\textstyle`, `\scriptstyle` 或者 `\scriptscriptstyle`; 所选定的字体将应用到公式或子公式结束, 或者直到你给出另外一种字体。例如, ‘`$$n+\scriptstyle n+\scriptscriptstyle n.$$`’得到的所有列表公式

$$n+\scriptstyle n+\scriptscriptstyle n.$$

这个例子比较蠢, 但是可以看到, 随着字体的改变, 加号也变得更小了。在标号字体中, TEX 不在 + 号两边添加间距。

下面是利用字体变化的一个更好的例子: 有时候需要输入“连分数”, 由许多其它分数组成, 所有的都被

假定用列表字体:

$$a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\cfrac{1}{a_3+\cfrac{1}{a_4}}}}$$

为了得到这种效果, 要输入的是:

```
$$a_0+{1\over\displaystyle a_1+
        {\strut 1\over\displaystyle a_2+
          {\strut 1\over\displaystyle a_3+
            {\strut 1\over a_4}}}}$$
```

控制系列 `\strut` 被用来使分母更高; 这是第十八章要讨论的微调。我们现在关心的是字体命令。) 如果在此公式中不出现 `\strut` 和 `\displaystyle`, 那么结果将完全不同:

$$a_0+{1\over a_1+{1\over a_2+{1\over a_3+{1\over a_4}}}}$$

这些例子表明, 分数中的分子和分母一般都相对居中。如果要分子或分母左对齐, 就在其后加上'`\hfill`'; 如果要右对齐, 就在左边加上'`\hfill`'。例如, 如果前一个例子中前三个'`1\over`'用'`1\hfill\over`'代替, 得到的列表公式为:

$$a_0+\cfrac[l]{1}{a_1+\cfrac[l]{1}{a_2+\cfrac[l]{1}{a_3+\cfrac{1}{a_4}}}}$$

(这种连分数的格式可能是许多作者想要的)。 之所以能这样的原因是, `\hfill` 的伸长能力比分子和分母居中时在内部实际所使用的粘连更大。

TEX 有另外一个命令'`\atop`', 它象 `\over`, 但是没有分数中的横线:

`$$x\atop y+2$$` $${x\atop y+2}$$

在附录 B 的 plain TEX 格式中还定义了'`\choose`', 它象 `\atop`, 但是把结果封装在括号中了:

`$$n\choose k$$` $${n\choose k}$$

之所以叫做 `\choose` 是因为它就是所谓二项式系数的通用符号, 给出了从 n 个中间取 k 个有多少种方法。

不要把 `\over` 和 `\atop` 和 `\choose` 混合使用。例如, '`$$n \choose k \over 2$$`'是不对的; 你必须使用编组, 输入'`$${n\choose k}\over2$$`'或者'`$$n\choose{k\over2}$$`', 即,

$${{n\choose k}\over 2} \qquad 或 \qquad {n\choose{k\over2}}.$$

顺便说一下, 后一个公式最好写作‘`$$n\choose k/2$$`’或‘`$$n\choose{1\over2}k$$`’, 其结果是

$$\binom{n}{k/2} \qquad \text{or} \qquad \binom{n}{\frac12 k}.$$

▶ **练习17.6**

作为 $\frac{\binom{n}{k}}{2}$ 的另一种方法, 看看怎样才能得到两个列表公式

$$\frac12\binom{n}{k} \qquad \text{和} \qquad \frac{\displaystyle\binom{n}{k}}{2}.$$

▶ **练习17.7**

看看怎样得到列表公式

$$\binom{p}{2}x^2y^{p-2}-\frac{1}{1-x}\frac{1}{1-x^2}.$$

 TEX 还有一个比的 `\over` 和 `\atop` 更灵活命令, 用它可以准确地给出横线的粗细, 只要输入‘`\above`⟨dimen⟩’即可。例如,

```
$$\displaystyle{a\over b}\above1pt\displaystyle{c\over d}$$
```

将得到一个复合分数, 其主横线更粗(1 pt):

$$\frac{\displaystyle\frac{a}{b}}{\displaystyle\frac{c}{d}}.$$

这种东西主要出现在初等数学的教科书中。

数学家通常用符号 $\sum$ 来表示“求和”, 用符号 $\int$ 表示“积分”。如果你只是一个排版者而不是数学家, 就只需要记住 `\sum` 表示 $\sum$ 和 `\int` 表示 $\int$; 如果你忘记了, 这些简写与其它所有符号都在附录 F 中。象 $\sum$ 和 $\int$ 这样的符号(以及列在附录 F 中的象 $\bigcup$, $\prod$, $\oint$ 和 $\bigotimes$ 这样的其它符号)称为巨算符, 在输入时象输入普通符号或字母一样。差别在于, TEX 在列表字体中选择的巨算符比文本字体中要更大。例如,

`$\sum x_n$`	得到的是	$\sum x_n$	(字体 T)
`$$\sum x_n$$`	得到的是	$\displaystyle\sum x_n$	(字体 D).

列表公式的 `\sum` 通常伴有“上下限”, 即在它上面和下面有子公式。输入上下限就象输入上下标一样; 例如, 如果要得到

$$\sum_{n=1}^m$$

可以输入‘`$$\sum_{n=1}^m$$`’或‘`$$\sum^m_{n=1}$$`’。按照数学排版的正常约定, 如果出现在文本字体而不是列表字体中, TEX 就把它变成‘$\sum_{n=1}^m$’(即没有上下限了)。

积分与求和略有不同, 即使在列表字体中, 上下标也不变成上下限:

`$\int_{-\infty}^{+\infty}$` 得到的是 $\int_{-\infty}^{+\infty}$ (字体 T)

`$$\int_{-\infty}^{+\infty}$$` 得到的是 $\displaystyle\int_{-\infty}^{+\infty}$ (字体 D).

有些排版者希望在 $\int$ 上有上下限; 这要占用更多的页面, 但是如果子公式很复杂杂时, 这样做效果很好, 因为它把上下限从公式的其它内容区分开了。类似地, 有时候希望在文本字体或标号字体中使用上下限; 但是某些用户不要在列表公式的 $\sum$ 上出现上下限。直接在巨算符后面输入'`\limits`'或'`\nolimits`', 就可以改变 TEX 的约定。 例如:

`$$\int\limits_0^{\pi\over2}$$` 得到的是 $\displaystyle\int\limits_0^{\frac{\pi}{2}}$

`$$\sum\nolimits_{n=1}^m$$` 得到的是 $\displaystyle\sum\nolimits_{n=1}^m$

如果输入的是'`\nolimits\limits`'(大概是象 `\int` 这样的某些宏已经给出了 `\nolimits`, 但是你又想要上下限), 那么最后一个优先。还有一个命令'`\displaylimits`', 用它来恢复 TEX 的正常约定; 即上下限只在字体 D 和 D' 中出现。

有时候可能要在巨算符下面放两行或多行极限; 可以用'`\atop`'来实现。例如, 如果要得到列表公式

$$\sum_{\scriptstyle 0\le i\le m\atop\scriptstyle 0<j<n} P(i,j)$$

应该输入

```
$$\sum_{\scriptstyle0\le i\le m\atop\scriptstyle0<j<n}P(i,j)$$
```

(在文稿中可以用几个间距把它调整得更好看)。 命令'`\scriptstyle`'必须在这里出现两次——否则'$0\le i\le m$'和'$0<j<n$'就使用小标号尺寸, 那太小了。这是另一个 TEX 的自动规则应该改变的少见的例子。

▶ **练习17.8**

怎样输入列表公式 $\displaystyle\sum_{i=1}^p\sum_{j=1}^q\sum_{k=1}^r a_{ij}b_{jk}c_{ki}$?

▶ **练习17.9**

怎样得到 $\displaystyle\sum_{\scriptstyle 1\le i\le p \atop {\scriptstyle 1\le j\le q \atop \scriptstyle 1\le k\le r}} a_{ij}b_{jk}c_{ki}$?

因为数学公式可以大得惊人, 所以 TEX 必须可以生成不断增大的符号。例如, 如果输入

```
$$\sqrt{1+\sqrt{1+\sqrt{1+
          \sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+x}}}}}}}$$
```

就在结果中出现了各种用到的根号:

$$\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+x}}}}}}}$$

在这里, 最大的三个符号基本上是一样的, 除了垂直线段'$\vert$'必要地重复到所要求的尺寸外; 但是更小的符号是 TEX 的数学字体中的不同字符。

类似情况还出现在括号和其它所谓是"分界符"上。例如, 下面是 plain TEX 在公式中使用的各种尺寸的圆括号和大括号:

$$\Biggl(\Biggl(\Biggl(\Biggl(\biggl(\Bigl(\bigl((((\;)))\bigr)\Bigr)\biggr)\Biggr)\Biggr)\Biggr)\Biggr)\Biggl\{\Biggl\{\Biggl\{\Biggl\{\biggl\{\Bigl\{\bigl\{\{\{\{\;\}\}\}\bigr\}\Bigr\}\biggr\}\Biggr\}\Biggr\}\Biggr\}\Biggr\}$$

在每种情况下, 最大的三对都是通过重复扩展而得到的, 所以它们可以变得如所需要的那样大。

对数学家而言, 分界符很重要, 因为它们从外观上把复杂的公式的内在结构给理顺了; 它们把各个不同的子公式分开。下面给出 plain TEX 的 22 个基本分界符:

<table>
<tr><th>输入</th><th>分界符</th></tr>
<tr><td><code>(</code></td><td>left parenthesis: $($</td></tr>
<tr><td><code>)</code></td><td>right parenthesis: $)$</td></tr>
<tr><td><code>[</code> or <code>\lbrack</code></td><td>left bracket: $[$</td></tr>
<tr><td><code>]</code> or <code>\rbrack</code></td><td>right bracket: $]$</td></tr>
<tr><td><code>\{</code> or <code>\lbrace</code></td><td>left curly brace: $\{$</td></tr>
<tr><td><code>\}</code> or <code>\rbrace</code></td><td>right curly brace: $\}$</td></tr>
<tr><td><code>\lfloor</code></td><td>left floor bracket: $\lfloor$</td></tr>
<tr><td><code>\rfloor</code></td><td>right floor bracket: $\rfloor$</td></tr>
<tr><td><code>\lceil</code></td><td>left ceiling bracket: $\lceil$</td></tr>
<tr><td><code>\rceil</code></td><td>right ceiling bracket: $\rceil$</td></tr>
<tr><td><code>\langle</code></td><td>left angle bracket: $\langle$</td></tr>
<tr><td><code>\rangle</code></td><td>right angle bracket: $\rangle$</td></tr>
<tr><td><code>/</code></td><td>slash: $/$</td></tr>
<tr><td><code>\backslash</code></td><td>reverse slash: $\backslash$</td></tr>
<tr><td><code>|</code> or <code>\vert</code></td><td>vertical bar: $\vert$</td></tr>
<tr><td><code>\|</code> or <code>\Vert</code></td><td>double vertical bar: $\Vert$</td></tr>
<tr><td><code>\uparrow</code></td><td>upward arrow: $\uparrow$</td></tr>
<tr><td><code>\Uparrow</code></td><td>double upward arrow: $\Uparrow$</td></tr>
<tr><td><code>\downarrow</code></td><td>downward arrow: $\downarrow$</td></tr>
<tr><td><code>\Downarrow</code></td><td>double downward arrow: $\Downarrow$</td></tr>
</table>

`\updownarrow`	up-and-down arrow: $\updownarrow$
`\Updownarrow`	double up-and-down arrow: $\Updownarrow$

在某些情况下, 可以通过两种方法得到同一个分界符; 例如, 可以通过'`[`'或'`\lbrack`'得到左方括号。给出后一种方法是因为不是在所有的键盘上, 符号'`[`'都那样好用。但是要记住, 不要直接输入'`{`'或'`}`'来得到左或右大括号; 符号 `{` 和 `}` 已经保留给编组使用了。正确的方法是输入'`\{`'或'`\}`'或'`\lbrace`'或'`\rbrace`'。

要得到略大的任何这些符号, 只需要在它们前面加上'`\bigl`'(对开分界符)或'`\bigr`'(对闭分界符)。这使得包含多层分界符的公式容易阅读:

输入	输出
`$\bigl(x-s(x)\bigr)\bigl(y-s(y)\bigr)$`	$\bigl(x-s(x)\bigr)\bigl(y-s(y)\bigr)$
`$\bigl[x-s[x]\bigr]\bigl[y-s[y]\bigr]$`	$\bigl[x-s[x]\bigr]\bigl[y-s[y]\bigr]$
`$\bigl\| \|x\|+\|y\| \bigr\|$`	$\bigl\| \|x\|+\|y\| \bigr\|$
`$\bigl\lfloor\sqrt A\bigr\rfloor$`	$\bigl\lfloor\sqrt A\bigr\rfloor$

`\big` 分界符只比普通的要大得足以感觉到不同, 但是还是足够小得可在段落的文本中使用。这里是它们 22 个的全部, 为普通尺寸和 `\big` 尺寸:

$$()[]\{\}\lfloor\rfloor\lceil\rceil\langle\rangle/\backslash|\|\uparrow\Uparrow\downarrow\Downarrow\updownarrow\Updownarrow$$

$$\bigl(\bigr)\bigl[\bigr]\bigl\{\bigr\}\bigl\lfloor\bigr\rfloor\bigl\lceil\bigr\rceil\bigl\langle\bigr\rangle\big/\big\backslash\big|\big\|\big\uparrow\big\Uparrow\big\downarrow\big\Downarrow\big\updownarrow\big\Updownarrow$$

还可以通过 `\Bigl` 和 `\Bigr` 来得到列表公式中的适当大小的符号:

$$\Bigl(\Bigr)\Bigl[\Bigr]\Bigl\{\Bigr\}\Bigl\lfloor\Bigr\rfloor\Bigl\lceil\Bigr\rceil\Bigl\langle\Bigr\rangle\Big/\Big\backslash\Big|\Big\|\Big\uparrow\Big\Uparrow\Big\downarrow\Big\Downarrow\Big\updownarrow\Big\Updownarrow$$

它们比 `\big` 符号大50%。列表公式中最经常使用的分界符甚至更高(`\big` 尺寸的两倍); 这样的分界符由 `\biggl` 和 `\biggr` 构造, 它们看起来象:

$$\biggl(\biggr)\biggl[\biggr]\biggl\{\biggr\}\biggl\lfloor\biggr\rfloor\biggl\lceil\biggr\rceil\biggl\langle\biggr\rangle\bigg/\bigg\backslash\bigg|\bigg\|\bigg\uparrow\bigg\Uparrow\bigg\downarrow\bigg\Downarrow\bigg\updownarrow\bigg\Updownarrow$$

最后, `\Biggl` 和 `\Biggr` 的分界符是 `\bigl` 和 `\bigr` 的 2.5 倍:

$$\Biggl(\Biggr)\Biggl[\Biggr]\Biggl\{\Biggr\}\Biggl\lfloor\Biggr\rfloor\Biggl\lceil\Biggr\rceil\Biggl\langle\Biggr\rangle\Bigg/\Bigg\backslash\Bigg|\Bigg\|\Bigg\uparrow\Bigg\Uparrow\Bigg\downarrow\Bigg\Downarrow\Bigg\updownarrow\Bigg\Updownarrow$$

▶ **练习17.10**

看看怎样用列表字体输入公式 $\biggl(\dfrac{\partial^2}{\partial x^2}+\dfrac{\partial^2}{\partial y^2}\biggr)\bigl|\varphi(x+iy)\bigr|^2=0$, 使用 `\bigg` 分界符来得到大的括号。(符号 ∂ 和 φ 分别叫做 `\partial` 和 `\varphi`。)

▶ **练习17.11**

在实际使用中, `\big` 和 `\bigg` 分界符比 `\Big` 和 `\Bigg` 分界符更常用。想想为什么?

\bigl, \Bigl, \biggl 或 \Biggl 分界符是开符号, 象左圆括号一样; \bigr, \Bigr, \biggr 或 \Biggr 分界符是闭符号, 象右圆括号一样。Plain TEX 还给出了\bigm, \Bigm, \biggm 和 \Biggm 分界符, 它们用在公式中央; 这样的分界符起着表示关系的作用, 就象等号一样, 所以 TEX 在它两边都添加一点间距。

```
$\bigl(x\in A(n)\bigm|x\in B(n)\bigr)$
```
$\bigl(x\in A(n)\bigm|x\in B(n)\bigr)$

```
$\bigcup_n X_n\bigm\|\bigcap_n Y_n$
```
$\bigcup_n X_n\bigm\|\bigcap_n Y_n$

也可以只用 \big, \Big, \bigg 或 \Bigg; 它得到的分界符就一个普通变量一样。它主要用于斜线和反斜线, 就象下面的例子一样。

```
$${a+1\over b}\bigg/{c+1\over d}$$
```
$${a+1\over b}\bigg/{c+1\over d}$$

▶ **练习17.12**
用专业的方法输入 $\bigl(x+f(x)\bigr)\big/\bigl(x-f(x)\bigr)$? (看起来要不松散。)

TEX 有一个内置的机制, 能确定要封装住给定公式所需要的这对分界符的高度; 因此, 你可以使用这种方法, 而不在 \big, \bigg 或其它中筛选。所要做的只是

\left⟨delim$_1$⟩⟨subformula⟩\right⟨delim$_2$⟩

TEX 将排版此子公式, 把规定的分界符放在左右两边。分界符的大小正好可以框住子公式。例如, 在列表公式

```
$$1+\left(1\over1-x^2\right)^3$$
```
$$1+\left(1\over1-x^2\right)^3$$

中, TEX 选择了 \biggl(和 \biggr), 因为再小的话就框不住这个特殊的分数了。象'$\left(x\right)$'得到的就是'(x)'这样的简单公式, 因此, \left 和 \right 有时候选择的分界符比 \bigl 和 \bigr 更小。

只要用到 \left 和 \right, 它们就必须成对出现, 就象编组中的大括号一样。不能在一个公式中使用 \left 却在另一个中使用 \right, 也不能象'\left(...{...\right)...}'或'\left(...\begingroup...\right)...\endgroup'这样输入。这个限制的意义在于, 在 TEX 决定使用多大的分界符前, 需要把 \left 和 \right 之间的子公式排版完毕。但是这里值得提醒的是, 因为当你不使用 \left 和 \right 时, 不必配对圆括号和方括号等等这些: 如果输入象'$[0,1)$', '$)($'或者'$)$'这样的公式 TEX 不会认为是错误的。(TEX 这样做是对的, 因为这样不配对的公式太经常出现在数学文章中了。) 甚至当使用 \left 和 \right 时, TEX 也不要求配对的是以前出现的那个分界符; 因此, 只要你知道在做什么(甚至不知道也没关系), 就可以输入象'\left)'和/或'\right('这样奇怪的配对。

在上面的例子中, \over 没有作用在公式开头的'1+'上; 这是因为 \left 和 \right 有编组的功能, 而不仅仅是选择分界符大小。出现在 \left 和 \right 之间的任何定义都是局域的, 就象大括号出现在要封装的子公式外面一样。

▶ **练习17.13**

利用 `\left` 和 `\right` 输入下列列表公式(`\phi` 得到的是 ϕ):

$$\pi(n)=\sum_{k=2}^{n}\left\lfloor\frac{\phi(k)}{k-1}\right\rfloor.$$

这时, 你可能想知道, 既然 `\left` 和 `\right` 会自动算出大小, 为什么还要花精力学习 `\bigl` 和 `\bigr` 这些东西呢? 嗯, `\left` 和 `\right` 的确很方便, 但是至少在三种情况下你要凭自己的才智来得到正确的分界符尺寸: (1). 有时候 `\left` 和 `\right` 选择的比所要的小。例如, 在前面的一个示例中, 我们用 `\bigl` 和 `\bigr` 得到了$\bigl||x|+|y|\bigr|$; `\left` 和 `\right` 不能把分界符变得比所要求的更大, 因此'`$\left|\left|x\right|+\left|y\right|\right|$`' 得到的只是'$||x|+|y||$'。(2). 有时候 `\left` 和 `\right` 选择的比所要的大。当在列表公式中把一个巨算符封装起来时常常会出现这种情况; 例如, 比较下列两个公式:

```
$$\left( \sum_{k=1}^n A_k \right)$$
```

$$\left(\sum_{k=1}^n A_k\right)$$

```
$$\biggl( \sum_{k=1}^n A_k \biggr)$$
```

$$\biggl(\sum_{k=1}^n A_k\biggr)$$

`\left` 和 `\right` 的规则使得它们要把 `\sum` 以及上下限都封装起来, 但是在这样的特殊情况下, 最好让上下限露出一点; `\bigg` 分界符更好一些。(3). 有时候需要把一个大的列表公式分在两个或多个行, 并且知道开和闭分界符大小相同; 而你不能在第一行用 `\left` 并且在最后一行用 `\right`, 因为 `\left` 和 `\right` 必须配对出现。解决方法就是(比如说)在第一行用 `\Biggl` 并且在最后一行用 `\Biggr`。

当然, `\left` 和 `\right` 的一个优点就是可以得到任意大的分界符——比 `\biggggg` 还要大很多! 但是斜线和角括号却有最大尺寸; 如果要使用这些符号的大尺寸, 得到的是可用的最大尺寸。

▶ **练习17.14**

看看你掌握了分界符了吗: 强制 TEX 得到下列公式

$$\pi(n)=\sum_{m=2}^{n}\left\lfloor\left(\sum_{k=1}^{m-1}\bigl\lfloor(m/k)/\lceil m/k\rceil\bigr\rfloor\right)^{-1}\right\rfloor.$$

如果在 `\left` 和 `\right` 后面跟的是'`.`', 而不是所规定的基本分界符, 就得到所谓的空分界符(它是一个空白)。可能你想知道要这个有什么用。嗯, 有时候你要得到一个只包含一个巨分界符的公式。例如, 列表公式

$$|x|=\begin{cases}x, & \text{if } x\ge 0\\ -x, & \text{if } x<0\end{cases}$$

有'{'而没有'}'。可以通过下列方法得到它:

```
$$|x|=\left\{ ... \right.$$
```

第十八章讨论了'`...`'所代表的内容; 现在我们只需知道'`\right.`'在可见的左大括号后面产生一个不可见的右分界符。

空分界符不是完全置空的; 它是一个空盒子, 其宽度为 TEX 中一个叫 `\nulldelimiterspace` 的参数。后面我们将看到, 空分界符被插入到分数后面。Plain TEX 设置 `\nulldelimiterspace =1.2pt`。

当 TEX 要得到的是分界符时, 可以用 ‘`<`’ 或 ‘`>`’ 来代替 `\langle` 和 `\rangle`, 例如, ‘`\bigl<`’等价于‘`\bigl\langle`’, ‘`\right>`’等价于‘`\right\rangle`’。当然, ‘`<`’和‘`>`’一般得到的是小于和大于关系符号, 它们与角括号‘$\langle\,\rangle$’差别很大。

Plain TEX 还有几个分界符, 它们不在 22 个的基本集合中, 因为它们是特殊类型的。控制系列 `\arrowvert`, `\Arrowvert` 和 `\bracevert` 是通过分别重复垂直箭头, 垂直双箭头和巨括号的垂直部分而得到的。它们生成的结果类似于 `\vert` 或 `\Vert`, 但是它们两边要添加间距, 并且其厚度也不同。你也可以用 `\lgroup` 和 `\rgroup` 来得到没有中间部分的括号; 并且 `\lmoustache` 和 `\rmoustache` 将给出巨括号的顶部和底部字符。例如, 下面是 `\vert`, `\Vert` 以及七个特殊分界符的的 `\Big` 和 `\bigg` 尺寸:

$$\cdots\Big\vert\cdots\Big\Vert\cdots\Big\arrowvert\cdots\Big\Arrowvert\cdots\Big\bracevert\cdots\Big\lgroup\cdots\Big\rgroup\cdots\Big\lmoustache\cdots\Big\rmoustache\cdots;$$

$$\cdots\bigg\vert\cdots\bigg\Vert\cdots\bigg\arrowvert\cdots\bigg\Arrowvert\cdots\bigg\bracevert\cdots\bigg\lgroup\cdots\bigg\rgroup\cdots\bigg\lmoustache\cdots\bigg\rmoustache\cdots.$$

注意, `\lgroup` 和 `\rgroup` 非常类似与加粗的圆括号, 只是角上弯得更厉害; 这在大的列表公式中使用很好。但是不能把它们象圆括号那样使用, 因为它们只能用在大尺寸(`\Big` 或更大)。

问题: 如果巨分界符后面跟一个上下标时会出现什么情况? 回答: 问得好。在 `\left` 分界符后面, 它就是所封装子公式中的第一个上下标, 因此在它前面的是 `{}`。在 `\right` 分界符后面, 它就是整个 `\left...\right` 子公式的上下标。在 `\bigl`, `\bigr`, `\bigm` 或 `\big` 分界符后面, 它只是那个分界符的上下标。因此, ‘`\bigl(_2`’与‘`\left(_2`’得到的结果不同。

如果你仔细观察本章的数学排版, 就会注意到巨圆括号和大括号关于基线附近一条看不见的水平线对称; 当分界符变大时, 高度和深度增加同样的量。这个水平线称为公式的**轴**; 例如, 当前段落的文本中公式的轴为 ——。在每个分数中的横线都在轴上, 不管分子或分母的大小。

有时候必须生成一个关于轴垂直居中的特殊的盒子。(例如, 上面的‘$|x| = \{\ldots$’就用了这样的盒子。) TEX 提供了实现它的简单方法: 只要输入

`\vcenter{`⟨vertical mode material⟩`}`

垂直模式的内容就放在一个盒子中, 就象 `\vcenter` 是一个 `\vbox` 一样。接着盒子被升降到顶边和底边距轴一样的地方。

“轴”的概念只对数学公式有意义, 对普通文本没意义; 因此, TEX 只允许在数学模式下使用 `\vcenter`。如果的确需要在水平模式下垂直居中某些内容, 只能用‘`$\vcenter{...}$`’。(顺便说一下, 命令‘`\vcenter to`⟨dimen⟩’和‘`\vcenter spread`⟨dimen⟩’在数学模式下也是可以的; 垂直粘连总可以通过第十二章的规则来设置。但是一般直接用 `\vcenter` 就可以了。)

通过直接按照正常方法输入 \hbox, \vbox, \vtop, \box 或 \copy 就可以把任何盒子放在公式中, 即使是在数学模式下。还有, 可以象在水平模式下那样使用 \raise 或 \lower, 并且可以用 \vrule 插入垂直标尺。象 \vcenter 这样得到的盒子在数学公式中象普通符号那样使用。

当遇到字体中没有的不常用的符号时, 有时候需要制作你自己的符号。如果新符号只在一个地方使用, 就可以用 \hbox 或 \vcenter 或其它东西插入你想要的; 但是如果要定义一个普遍使用的宏, 那么可能要在不同字体中使用不同的命令。 TeX 提供了一种这种情况下使用的特殊命令, 叫做 \mathchoice: 给出

\mathchoice{⟨math⟩}{⟨math⟩}{⟨math⟩}{⟨math⟩}

其中每一个 ⟨math⟩ 规定了一个子公式。在字体 D 或 D' 中 TeX 选择第一个子公式, 在 T 或 T' 中选第二个, 在 S 或 S' 中选第三个, 在 SS 或 SS' 中选第四个。(实际上, 在 TeX 选定最后一个之前, 它把所有四个子公式都排版了, 因为在读入 \mathchoice 时一般还不知道实际要用的字体; 例如, 当使用'\over'时, 你常常要改变的是公式的前半部分出现所有公式的字体。因此, \mathchoice 要花费很多时间和空间, 因此只有在舍得的时候再使用它。)

▶ **练习17.15**

看看下列命令输出结果是什么:

```
\def\puzzle{{\mathchoice{D}{T}{S}{SS}}}
$$\puzzle{\puzzle\over\puzzle^{\puzzle^\puzzle}}$$
```

▶ **练习17.16**

设计一个叫'\square'的宏, 以得到数学公式中使用的盒子关于轴是对称的, 并且其内部尺寸在列表字体和文本字体中为 3 pt, 在标号字体中为 2.1 pt, 在小标号字体中为 1.5 pt。在列表字体和文本字体中为 0.4 pt 厚, 其它为 0.3 pt 厚。

Plain TeX 有一个叫 \mathpalette 的宏, 它用到命令 \mathchoice, '\mathpalette\a{xyz}'展开了四个分支 '\mathchoice{\a\displaystyle{xyz}}...{\a\scriptscriptstyle{xyz}}'。因此, \mathpalette 的第一个变量是一个控制系列, 而这个控制系列的第一个变量是选择字体。附录 B 包含了几个展示 \mathpalette 应用的例子。(特别是 \phantom, \root 和 \smash 的定义; 全等符号 \cong ($\cong$) 也是通过 \mathpalette 从 $=$ 和 $\sim$ 来得到的。

在本章开头, 我们讨论了命令 \over, \atop, \choose 和 \above。它们是 TeX 的广义分数特性的特殊情形, 这个广义分数还包括三个原始控制系列:

\overwithdelims⟨delim$_1$⟩⟨delim$_2$⟩
\atopwithdelims⟨delim$_1$⟩⟨delim$_2$⟩
\abovewithdelims⟨delim$_1$⟩⟨delim$_2$⟩⟨dimen⟩

其中的第三个更广义, 因为它包含了所有其它的广义分数: \overwithdelims 使用了厚度为当前尺寸的分数横线, \atopwithdelims 使用了厚度为零的看不见的分数横线, 而 \abovewithdelims 使用的是厚度为

任意给定的分数横线。TEX 直接把前面的子公式(分子)放在后面的子公式(分母)上面, 中间是所要求厚度的横线; 接着把 ⟨delim$_1$⟩ 放在左边, 把 ⟨delim$_2$⟩ 放在右边。例如, '`\choose`'等价于'`\atopwithdelims()`'。如果要定义 `\legendre` 为'`\overwithdelims()`', 那么可以通过输入'`{a\legendre b}`'来排版出 Legendre 符号'$\left(\frac{a}{b}\right)$'。两边的分界符大小与字体有关, 与分数的大小无关; 在字体 D 和 D' 中用大的分界符(见附录 G)。简单命令 `\over`, `\atop` 和 `\above` 等价于空分界符时'`withdelims`'的相应命令; 例如, '`\over`'就等价于'`\overwithdelims..`'。

▶ **练习17.17**
定义有关控制系列 `\euler`, 使得在公式中输入'`{n\euler k}`'时得到 Euler 数 $\left\langle{n \atop k}\right\rangle$。

附录 G 讨论了怎样精确算出 `\left` 和 `\right` 分界符所要的尺寸。一般思路是, 分界符关于轴垂直居中; 因此如果要框住 `\left` 和 `\right` 之间的子公式, 并且设轴上面的高度为 y_1, 轴下面的高度为 y_2, 那么我们要制作的分界符的高度加深度必须至少为 $y = 2\max(y_1, y_2)$。但是一般最好不完全框住公式, 而是仅仅基本上框住就可以了; 因此 TEX 允许给出两个参数, `\delimiterfactor` f(一个整数)和 `\delimitershortfall` δ (一个尺寸)。分界符的最小尺寸至少取为 $y \cdot f/1000$, 并且至少为 $y - \delta$。附录 B 设置 $f = 901$ 和 $\delta = 5\,\mathrm{pt}$。因此, 如果 $y = 30\,\mathrm{pt}$, 那么 plain TEX 公式就把分界符的高度变成 27 pt; 如果 $y = 100\,\mathrm{pt}$, 相应的分界符的高度至少为 95 pt。

现在我们已经讨论了排版数学公式的规则, 但是关于 TEX 怎样把输入内容变成盒子和粘连列还没有讲述。在第十六和十七章提到的几乎所有控制系列都是 plain TEX 格式的"高级"命令; 它们不是 TEX 自己内建的。附录 B 用 TEX 实际使用的原始命令定义了这些控制系列。例如, '`\choose`'的定义为'`\atopwithdelims()`'; 附录 B 不仅定义了 `\choose`, 它还告诉 TEX 怎样得到各种大小的分界符 (和)。Plain TEX 格式定义了所有象 `\alpha` 和 `\mapsto` 这样的特殊字符, 所有象 `\tilde` 和 `\widehat` 这样的特殊重音, 所有象 `\sum` 和 `\int` 这样的巨算符, 所有象 `\lfloor` 和 `\vert` 这样的分界符。这些东西的任一个都可重新定义, 这样的话 TEX 就可以使用其它数学字体和或其它字体了。

本章剩下的内容要讨论 TEX 在幕后实际使用的低级命令。在下几页的每段前都有两个"危险"标识, 因此除非特别着迷, 你可以跳过第十八章了。

在数学模式下排版的所有字符都属于 16 个字体族中的一个, 在内部用从 0 到 15 来编号。这些族中的每个都包含三种字体: 一个用于文本尺寸, 一个用于标号尺寸, 一个用于小标号尺寸。命令 `\textfont`, `\scriptfont` 和 `\scriptscriptfont` 对应于每族的成员。例如, plain TEX 格式中的第 0 族被用于 roman 字母, 并且附录 B 用下列指令

```
\textfont0=\tenrm
\scriptfont0=\sevenrm
\scriptscriptfont0=\fiverm
```

来设定此族: 10-point roman 字体 (`\tenrm`) 用于正常符号, 7-point roman 字体 (`\sevenrm`) 用于上下标, 5-point roman 字体 (`\fiverm`) 用于小上下标。因为每种字体有 256 个字符, 每族有 3 种字体, 总共有 16 族, 那么在任一公式中, TEX 都可直接得到 12,228 个字符(每种尺寸为 4096 个)。想像一下有多少。

象 \textfont⟨family number⟩=⟨font identifier⟩ 这样的定义对包含它的编组而言是局域的, 因此可以很容易在族成员之间从一组设定改为另一组并且再改回来。还有, 可以把任何字体放在任何族中; 例如, 命令

```
\scriptscriptfont0=\scriptfont0
```

把第 0 族小标号尺寸变得同当前的标号尺寸一样大。 TEX 不其检验族是否建立起来了; 它只遵循指令。(但是, 我们后面将看到, 字体不能用在第 2 和 3 族, 除非它们包含一定数量的参数。) 顺便说一下, 对没有定义的每族成员, TEX 都用不包含字符的 \nullfont 代替。

在数学公式正在读入期间, TEX 把每个符号记作"在某某族中的某某字符位置", 但是直到公式结尾, 它才去注意族中实际是什么字体。因此, 如果你已经把叫做 \Helvetica (它包含瑞士数字)的字体载入, 并且给出象

```
$\textfont0=\tenrm 9 \textfont0=\Helvetica 9$
```

这样的指令, 那么就得到字体 \Helvetica 中的两个 9, 如果 TEX 设定得要从第 0 族得到 9 的话。原因是在公式结尾处 \textfont0 是 \Helvetica, 并且那是它正在起作用的时候。另一方面, 如果使用

```
$\textfont0=\tenrm 9 \hbox{$9\textfont0=\Helvetica$}$
```

那么第一个 9 是 \tenrm 而第二个是 \Helvetica, 因为 hbox 中的公式在合并到外面公式前就已经排版好了。

▶ **练习17.18**
如果输入'${\textfont0=\Helvetica 9}$', 那么 9 所用的字体是什么?

每个数学字符被指定一个识别码数字, 在 0 和 4095 之间, 它等于 256 乘以族数再加上位置数。这很容易用十六进制表示, 一位十六进制数为族数, 两位为字符位置; 例如, "24A 表示第 2 族的 "4A 字符。每个字符还可以指定到 8 类中的一类, 编号从 1 到 7, 如下:

类	意思	例子	类	意思	例子
0	Ordinary(普通符号)	/	4	Opening(开符号)	(
1	Large operator(巨符号)	\sum	5	Closing(闭符号)	)
2	Binary operation(二元运算)	+	6	Punctuation(标点)	,
3	Relation(关系符号)	=	7	Variable family(变量族)	x

第 0 到 6 类是数学排版中字符属于哪个"讨论过的部分"; 第 7 类是下面要讨论的特殊情形。这些类数乘以 4096 再加到字符代码上, 而且这个类数就是就是四位十六进制数的第一位数。例如, 附录 B 把 \sum 定义为数学字符 "1350, 意思是它是巨算符(第 1 类), 在第 3 类的位置 "50 上。

▶ **练习17.19**
符号 \oplus 和 \bullet (⊕ 和 •)是二元运算, 在 plain TEX 的字体中, 出现在第 2 族位置 8 和 15 (十进制数)上。其数学字符代码是什么?(这太简单了。)

第 7 类是一种特殊情形, 它允许数学符号改变族数。它的性质象第 0 类, 但是给出的族数被叫做 \fam 的整数参数的当前值所代替, 只要 \fam 是一个合理的族数(即处在 0 和 15 之间)。只要进入了数学模式, TEX 就自动设置 \fam=-1; 因此, 如果 \fam 没有得到新的值, 那么第 7 类与第 0 类就是等价的。当用户输入'\rm'时, plain TEX 就把 \fam 变为零; 这样就很容易得到 roman 字母, 就象我们在第十八章将看到的那样, 因为字母属于第 7 类。控制系列 \rm 的定义为'\fam=0 \tenrm'; 因此, \rm 使 \fam 变成零, 并且它把"当前字体"变成 \tenrm。在水平模式下, \fam 的值无关紧要, 并且当前字体决定字母的排版; 但是在数学模式下, 当前字体是无关紧要的, \fam 的值决定字母的排版。只有当用到 \␣ 或要用 ex 或 \em 的方式给出尺寸时, 才在数学模式下用到当前字体; 如果 hbox 出现在公式中, 当前字体也起作用, 因为 hbox 的内容要在水平模式下排版。

数学模式下字符的含义由 256 个"mathcode"的值这个表来定义; 这些表的单元可以用命令 \mathcode 来改变, 就象类代码可以用 \catcode 改变一样(见第七章)。每个 mathcode 规定了类, 族和字符位置, 就象上面讲述的那样。例如, 附录 B 中含有命令

```
\mathcode‘<="313C
\mathcode‘*="2203
```

它告诉 TEX , 在数学模式下, 把字符'<'看作关系符号(第 3 类), 在第 1 族的位置 ″3C 上, 把星号'*'看作二元运算, 在第 2 族的位置 3 上。\mathcode‘b 的初始值为 ″7162; 因此 b 是第 1 族(italic)的字符 ″62, 并且其族要随着 \fam 变化。(对不是字母和数字的所有字符, INITEX 开始设置为 \mathcode $x = x$。10 个数字为 \mathcode $x = x +$ ″7000; 52 个字母为 \mathcode $x = x +$ ″7100。) 只有当 TEX 排版类代码为 11 (字母) 或 12 (其它)的字符时, 或者当读入的字符由 \char⟨number⟩ 给出时, 它才用到 mathcode。

\mathcode 也可以有一个特殊值 ″8000, 它使得字符的性质同类代码 13 (活动符)一样。附录 B 中利用这个性质略施小计, 把 ' 定义为 {\prime}。' 的 mathcode 不影响用八进制常数来使用 '。

mathcode 允许你把单个键间接地指向任何族中任何字符。你还可以通过输入 \mathchar 直接给出数学字符, 它类似于 \char。例如, 命令'\mathchar"1ABC'给出了第 1 类, 第 10 (″A) 族, 位置 ″BC 上的字符。因此象

```
\def\sum{\mathchar"1350 }
```

这样大概 100 个命令足以定义完 plain TEX 中的特殊符号了。但是, 有一个更好的方法: TEX 有一个原始命令 \mathchardef, 它与 \mathchar 的关系就象 \chardef 与 \char 的一样。附录 B 给出了象

```
\mathchardef\sum="1350
```

这样的 100 个左右的定义来定义特殊符号。\mathchar 必须在 0 和 32767 (″7FFF) 之间。

当排版第 1 类字符, 即象 \sum 这样的巨算符时, 它应该关于轴垂直居中。因此, 巨算符可以使用不同的字体尺寸。这个垂直调整对其它类的符号是没有的。

TEX 把类型与子公式以及单个字符联系起来。因此, 例如, 如果需要的话, 可以把一个复杂的式子看作二元运算或关系符号等等。为此要用到命令 \mathord, \mathop, \mathbin, \mathrel,

\mathopen, \mathclose 和 \mathpunct; 它们的每个后面都可以跟单个字符或者放在大括号中的子公式。例如, \mathopen\mathchar"1234 等价于 \mathchar"4234, 因为 \mathopen 把类强制变成第 4 类(开符号)。在公式‘$G\mathbin:H$’中, 冒号被看作二元运算。通过把 \bigl#1 定义为

```
\mathopen{\hbox{$\left#1 ...\right.$}}
```

附录 B 构造了大量开符号。还有第 8 类, \mathinner, 它一般不用在单个符号上; 分数或 \left...\right 式子被看作“内部”子公式时, 就意味着在某些时候它们外面要有额外的间距。所有其它子公式一般被看作普通符号, 不管它们是由 \overline, \hbox 或 \vcenter 生成的, 或者是封装在大括号中的。因此, \mathord 其实在 TEX 语言中是不必要的; 不用输入‘$1\mathord,234$’, 照样可以用‘$1{,}234$’得到同样的效果。

▶ **练习17.20**
象 \mathchardef\alpha="010B 这样的命令在附录 B 中用来定义小写希腊字母。假定你要扩展 plain TEX, 把加粗的数学 italic 字母放在第 9 族, 就象常用的数学 italic 字母放在第 1 族一样。(这样的字体在 TEX 最简版中不能用, 但是外面假定它存在。) 假设控制系列 \bmit 已经定义为‘\fam=9’; 因此, ‘{\bmit b}’将给出加粗的数学 italic b。怎样改变定义使得 {\bmit\alpha} 得到的是加粗的 alpha?

分界符由一种类似但更复杂的方法给出。每个字符不但有一个 \catcode 和一个 \mathcode, 还有一个 \delcode, 它或者是负值(对不作为分界符使用的字符), 或者小于 "1000000。换句话说, 非负的 delcode 由 6 位十六进制数组成。前三个规定了分界符的“小”组分, 后三个规定了“大”组分。例如, 命令

```
\delcode`x="123456
```

意思是, 如果字母 x 被作为分界符使用, 它的小组分在第 1 族的位置 "23 上, 大组分在第 4 族的位置 "56 上。但是如果大组分或小组分为 000 (第 0 族的位置 0 上), 此组分将忽略掉。当组分跟在 \left 或 \right 后面, 或者在某个 withdelims 命令后时, TEX 才用到 delcode; 负的 delcode 会产生一个错误信息, 但是其它时候 TEX 通过先找小组分再找大组分而得到一个适当的分界符。(附录 G 讨论了这个过程的细节。) 例如, 附录 B 中包含命令

```
\delcode`(="028300  \delcode`.=0
```

它表明, 左圆括号的小组分在第 0 族的位置 "28 上, 大组分在第 3 族的位置 0 上; 还有, 句点没有什么组分, 因此‘\left.’得到的是一个空分界符。实际上, 在第 3 族中有几个不同的左括号符号; 最小的在位置 0, 其它的通过字体中的信息链接在一起。所有的 delcode 在被命令 \delcode 改变之前都是 −1。

▶ **练习17.21**
附录 B 定义了 \delcode`<, 使得角括号有已经简短的名称。想想为什么附录 B 不进一步定义出\delcode`{?

分界符也可以直接给出, 就象‘\delimiter⟨number⟩’这样。在这种情况下, 数字最大为 "7FFFFFF, 即七位十六进制数; 领头的数字给出类, 从 0 到 7, 象在 \mathchar 中一样。例如, 附录 B 包含定

义

```
\def\langle{\delimiter"426830A }
```

它的意思是, \langle 是一个开符号(第 4 类), 其小组分为 ″268, 大组分为 ″30A。当 \delimiter 出现在 \left 或 \right 后面时, 类的数字忽略掉; 但是当 \delimiter 出现在其它情况下时, 即, 当 TEX 不把它看作分界符时, 三个最右边的数字去掉, 剩下的四个作为 \mathchar 出现。例如, 式子'$\langle x$'将被看作'$\mathchar"4268 x$'。

▶ **练习17.22**
输入'\bigl\delimiter"426830A'会产生什么错误?

当然 \mathchar 和 \delimiter 的这些数字约定不好看, 它们确实太臃肿了。这就是 TEX 把它们作为格式中低级定义的原因。还有两个低级原始命令要提一下: \radical 和 \mathaccent。Plain TEX 通过命令

```
\def\sqrt{\radical"270370 }
\def\widehat{\mathaccent"362 }
```

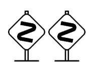
给出根号符号和数学重音, 还有几个类似的命令。思路是, \radical 后面跟的是分界符代码, \mathaccent 后面跟的是数学组分代码, 使得 TEX 知道了用在根号和重音构造中的族和字符位置。附录 G 给出了这些字符位置的准确信息。通过改变定义, TEX 很容易得扩展到排版各种不同的根号和各种不同的重音, 只要这样的符号在字体中可用。

Plain TEX 把第 1 族用于数学 italic 字母, 第 2 族用于普通数学符号, 第 3 族用于巨符号。 TEX 的第 2 和 3 族中给出了一个叫 \fontdimen 的特殊参数, 由它按照附录 G 的规则来控制数学间距; 符号字体 cmsy 和 cmex 有这些参数, 因此, 差不多把它们都指定到第 2 和 3 族中。(但是, 用命令 \fontdimen 可用修改任何字体的参数。) INITEX 初始化所有字母 A 到 Z 和 a 到 z 的 mathcode, 使得它们变成第 7 类第 1 族的符号; 这就是把第 1 族用于数学 italic 的原因。类似地, 数字 0 到 9 是第 7 类第 0 族。 TEX 不把其它族进行特殊对待了。因此, 例如, plain TEX 把文本 italic 字体放在第 4 族, slanted roman 字体放在第 5 族, bold roman 放在第 6 族, typewriter 字体放在第 7 族, 但是这些数字可以转换。有一个叫 \newfam 的宏, 类似于 \newbox, 它把符号的名称指向未使用的族。

当 TEX 在水平模式时, 它制作一个水平列; 在垂直模式下, 制作一个垂直列。因此, 在数学模式下当然得到的是数学列了。水平列的讨论见第十四章, 垂直列的讨论见第十五章; 现在讨论的是数学列。数学列中的每个项目都是下列某个类型:

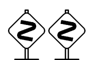
- 原子(马上就讨论);
- 水平内容(标尺, 可断点, penalty 或"whatsit");
- 垂直内容(来自 \mark, \insert 或 \vadjust);
- 粘连团(来自 \hskip, \mskip 或 \nonscript);
- kern (来自 \kern 或 \mkern);
- 字体改变(来自 \displaystyle, \textstyle, 等等);

- 广义分数(来自 \above, \over, 等等);
- 分界符(来自 \left 或 \right);
- 四分支选择(来自 \mathchoice).


最重要的项目称为原子, 有三个部分: 核, 上标, 下标。例如, 在数学模式下, 如果输入

```
(x_i+y)^{\overline{n+1}}
```

就得到由五个原子组成的数学列: (, x_i, +, y 和 $)^{\overline{n+1}}$。这些原子的核为 (, x, +, y 和); 除了第二项下标为 i 外其它下标都是空的; 除了最后一项的上标为 $\overline{n+1}$ 其它上标都是空的。这个上标自己也是由一个原子组成的数学列, 其核为 $n+1$; 这个核是由三个原子组成的数学列。


有十三种原子, 每种在公式中的表现都不同; 例如, '('是一个开原子, 因为它来自开符号。下面是不同种类的完整列表:

Ord	普通原子, 如'x';
Op	巨原子, 如'$\sum$';
Bin	二元运算原子, 如'$+$';
Rel	关系原子, 如'$=$';
Open	开原子, 如'(';
Close	闭原子, 如')';
Punct	标点原子, 如',';
Inner	内部原子, 如'$\frac{1}{2}$';
Over	上划线原子, 如'$\overline{x}$';
Under	下划线原子, 如'$\underline{x}$';
Acc	重音原子, 如'$\hat{x}$';
Rad	根号原子, 如'$\sqrt{2}$';
Vcent	\vcenter 生成的垂直居中 vbox。


原子的核, 上标和下标称为字段, 并且每个字段有四种可能; 一个字段可以是

- 空的;
- 数学符号(由族和位置给出);
- 盒子; 或者
- 数学列。

例如, 上面讨论的闭原子 $)^{\overline{n+1}}$ 的上标字段是空的; 核为符号')', 如果采用 plain TEX 的约定, 那么它是第 0 族的字符 "28; 其上标字段为数学列 $\overline{n+1}$。后一个数学列由一个上划线原子组成, 其核为数学列 $n+1$; 而这个数学列由三个原子组成, 分别是普通原子, 二元运算原子, 普通原子。

可以通过在数学模式下输入 `\showlists` 看看 TEX 是怎样处理的。例如, 编译‘`$(x_i+y)^{\overline{n+1}}\showlists$`’后, 在 log 文件中得到下列古怪的信息:

```
\mathopen
.\fam0 (
\mathord
.\fam1 x
_\fam1 i
\mathbin
.\fam0 +
\mathord
.\fam1 y
\mathclose
.\fam0 )
^\overline
^.\mathord
^..\fam1 n
^.\mathbin
^..\fam0 +
^.\mathord
^..\fam0 1
```

依照我们前面关于 `\showlists` 的经验可以看出, 有盒子套盒子, 而且 log 文件中每行前面的点表明它所处的层次。数学列的结构略微复杂一些; 因此, 一个点表示原子的核, 一个‘`^`’用来表示上标字段, 一个‘`_`’用来表示下标字段。空字段没有显示。因此, 例如, 普通原子 x_i 在这里用三行来表示: ‘`\mathord`’, ‘`.\fam1 x`’和‘`_\fam1 i`’。

某些种类的原子除了核, 上标和下标字段外还有额外的东西: 在正常的 `\displaylimits` 被覆盖时, 巨算符原子要标记上‘`\limits`’或‘`\nolimits`’; 根号原子包含根号要用到的分界符字段; 重音原子包含重音符号的族和字符代码。

当在数学模式下输入 `\hbox{...}` 时, 普通原子就放在了当前数学列上, 其中 hbox 为其核。类似地, `\vcenter{...}` 得到的 Vcent 原子的核是一个盒子。但是在大多数情况下, 原子的核是一个符号或一个数学列。可以用 `\showlists` 试试象分数和 mathchoice 在内部表示为什么东西了。

第二十六章包含了构造数学列的完整细节。一旦数学列结束了(即闭符号‘`$`’出现了), TEX 将卸开当前数学列, 并且把它转换到水平列。这个转换的规则在附录 G 给出。可以对比这样的数学排版的“前后”表示: 在公式结尾输入‘`\showlists$\showlists`’; 第一个 `\showlists` 显示的是数学列, 第二个显示的是从它加工出的水平列(可能很复杂)。

18. 精致的数学排版

我们已经讨论了大多数构造数学公式的工具, 但是还有几种东西, 好的数学排版者非看不可。在用第十六和第十七章的基本方法排版大约一打公式后, 就会发现当输入数学式子时, 就能想像出它的结果来。而且一旦你达到这种水平, 离你排版出世界上所见过的最漂亮的公式只差一点了; 适当地用 TEX 的技巧进行润色会给你排版的书籍和文章在观感和阅读上涂上一层专业的光泽。本章就讨论这些技巧, 而且还弥补了几个缺陷, 它们都是在第十六和第十七章中未完全解决的数学排版问题。

1. *标点* 当公式后面跟着一个句点, 逗号, 分号, 冒号, 问号, 惊叹号等等时, 且公式出现在文本中时, 把标点放在 `$` *后面*; 当公式是列表时, 把标点放在 `$$` *之前*。例如,

```
If $x<0$, we have shown that $$y=f(x).$$
```

当标点符号不看作公式的一部分时, 段落中 TEX 的间距规则处理得很好。

类似地, 不要输入象下面这样的方式:

```
for $x = a, b$, or $c$.
```

应该输入

```
for $x = a$, $b$, or $c$.
```

(要更好的话, 用带子: ‘`or~$c$`’。) 原因是, TEX 将把式子‘`$x = a, b$`’看作单个公式来排版, 这样在逗号和 b 之间的是“细间距(thin space)”。这个间距与 TEX 放在 b *后面*的逗号后的间距不一样, 因为单词间的间距总比细间距要大。如此不等的间距不好看, 但是当正确输入时, 间距看起来就舒服了。

不输入‘`$x = a, b$`’的另一个原因是, 它不允许在段落中被断行: TEX 从不在逗号和 b 之间的间距处断行, 因为在公式中的逗号后断行一般都是不对的。例如, 在方程‘`$x = f(a, b)$`’中, 我们当然不希望把‘$x = f(a,$’放在一行, 而把‘$b)$’放在下一行。

因此, 在段落的文本中输入公式时, 要准确地插入数学式子: 不要把象 $-$ 和 $=$ 这样的符号放在 `$` 外面, 而且如果逗号真是公式的一部分, 那么要放在公式中。但是如果逗号或句点或其它标点符号在语句上属于句子而不是公式, 要把它放在 `$` 外面。

▶ **练习18.1**
输入: $R(n,t) = O(t^{n/2})$, as $t \to 0^+$。

☡ 有些数学文体在公式两边插入一点额外间距从而把它们从文本中分离出来。例如, 当用无 italic 字母的普通打字机排版时, 高超的排版者一般在每个公式前后都要放一个额外的空格, 因为这带来了视觉上的不同。可能把每个 `$` 看作输出时的一个潜在额外间距是有好处的; 这样, 把句子的标点放在公式外面这个规则就好记住了。

☡☡ 实际上, TEX 的确在每个公式前后插入了额外间距; 这个间距的量叫做 `\mathsurround`, 它是一个值为 ⟨dimen⟩ 的参数。例如, 如果你设置 `\mathsurround=1pt`, 那么每个公式的宽度将增加

2 point(一边 1 pt):

For $x = a$, b, or c. (`\mathsurround=1pt`)
For $x = a, b,$ or c. (`\mathsurround=0pt`)

如果公式出现在行首或行尾, 那么这个额外的间距将融入左页边或右页边。当 TEX 读入公式的闭符号 `$` 时, 起作用的 `\mathsurround` 值就插入到此公式的左右两边。Plain TEX 设置 `\mathsurround=0pt`, 因此如果不使用别的格式, 或者自己不改变 `\mathsurround`, 就不会看到额外间距。

2. *公式中的非 italic 字母* 数学变量的名称一般是 italic 或希腊字母, 但是普通数学函数如'log'总设定为 roman 字体。处理这个情况的最好方法是使用下列 32 个控制系列(它们都定义在 plain TEX 格式中, 见附录 B):

```
\arccos  \cos   \csc  \exp  \ker     \limsup  \min  \sinh
\arcsin  \cosh  \deg  \gcd  \lg      \ln      \Pr   \sup
\arctan  \cot   \det  \hom  \lim     \log     \sec  \tan
\arg     \coth  \dim  \inf  \liminf  \max     \sin  \tanh
```

这些控制系列得到的是 roman 字体以及相应的间距:

输入	输出
`$\sin2\theta=2\sin\theta\cos\theta$`	$\sin2\theta=2\sin\theta\cos\theta$
`$O(n\log n\log\log n)$`	$O(n\log n\log\log n)$
`$\Pr(X>x)=\exp(-x/\mu)$`	$\Pr(X>x)=\exp(-x/\mu)$
`$$\max_{1\le n\le m}\log_2P_n$$`	$\max\limits_{1\le n\le m}\log_2P_n$
`$$\lim_{x\to0}{\sin x\over x}=1$$`	$\lim\limits_{x\to0}{\sin x\over x}=1$

最后两个公式是列表公式, 它们表明, 某些特殊控制系列被 TEX 看作"巨算符", 其上下限象 $\sum$ 的一样: `\max` 上的下标处理方式与 `\log` 的下标不同。在列表字体中, 当遇到 `\det`, `\gcd`, `\inf`, `\lim`, `\liminf`, `\limsup`, `\max`, `\min`, `\Pr` 和 `\sup` 时, 上下标将变成上下限。

▶ **练习18.2**

用 plain TEX 给出下列列表公式, 其中'ν'由'`\nu`'得到:

$$p_1(n)=\lim_{m\to\infty}\sum_{\nu=0}^\infty\bigl(1-\cos^{2m}(\nu!^n\pi/n)\bigr).$$


如果需要没包括在 plain TEX 的 32 个函数控制系列中的一些数学函数或算符的 roman 字体, 可以模仿附录 B 来定义新的控制系列。或者, 如果所需的 roman 字体只要用一次, 可以如下转换到 `\rm` 字体来得到它:

`$\sqrt{{\rm Var}(X)}$`	$\sqrt{{\rm Var}(X)}$

```
$x_{\rm max}-x_{\rm min}$
```
$x_{\max}-x_{\min}$

```
${\rm LL}(k)\Rightarrow{\rm LR}(k)$
```
${\rm LL}(k)\Rightarrow{\rm LR}(k)$

```
$\exp(x+{\rm constant})$
```
$\exp(x+{\rm constant})$

```
$x^3+{\rm lower\ order\ terms}$
```
$x^3+{\rm lower\ order\ terms}$

注意, 在最后一种情况下, 用到了'\␣'; 如果没有它们, 得到的将是'$x^3+{\rm lowerorderterms}$', 因为在数学模式下将忽略调普通空格。

在公式中, 还可以用 `\hbox` 而不是 `\rm` 来得到 roman 字母。例如, 最后五个公式中的四个也可以如下得到:

```
$\sqrt{\hbox{Var}(X)}$
```
$\sqrt{\hbox{Var}(X)}$

```
$\hbox{LL}(k)\Rightarrow\hbox{LR}(k)$
```
$\hbox{LL}(k)\Rightarrow\hbox{LR}(k)$

```
$\exp(x+\hbox{constant})$
```
$\exp(x+\hbox{constant})$

```
$x^3+\hbox{lower order terms}$
```
$x^3+\hbox{lower order terms}$

在这种情况下, 就不需要'\␣'了, 因为在 `\hbox` 中的内容是在水平模式下处理的, 这时空格是起作用的。但是这样使用 `\hbox` 有两个缺点: (1). 盒子中的内容按同一尺寸排版, 不管盒子是不是下标; 例如, '`$x_{\hbox{max}}$`'得到的是'$x_{\hbox{max}}$'。(2). 在盒子中使用的字体是"当前字体", 因此可能不是 roman 字体。例如, 如果正在用 slanted 字体陈述某些定理, 并且定理中出现了'`$\sqrt{\hbox{Var}(X)}$`', 那么就得到不想要的结果'$\sqrt{\textit{Var}(X)}$'。为了确保 `\hbox` 使用的是 roman 字体, 就要给出 `\rm`, 即'`$\sqrt{\hbox{\rm Var}(X)}$`'; 因此 `\hbox` 就没什么用处了。但是外面后面将看到, 在列表公式中 `\hbox` 非常有用。

▶ **练习18.3**

当用 plain TEX 标准宏时, 列表公式 '`$$\lim_{n\to\infty}x_n {\rm\ exists} \iff \limsup_{n\to\infty}x_n = \liminf_{n\to\infty}x_n.$$`'排版得到

$$\lim_{n\to\infty}x_n {\rm\ exists} \iff \limsup_{n\to\infty}x_n = \liminf_{n\to\infty}x_n.$$

但是有些人喜欢用不同的符号: 看看怎样改变 `\limsup` 和 `\liminf` 的定义, 使得结果为

$$\lim_{n\to\infty}x_n {\rm\ exists} \iff \overline{\lim_{n\to\infty}}\,x_n = \underline{\lim_{n\to\infty}}\,x_n.$$

词'mod'一般也在公式中设定为 roman 字体; 但是这个词需要更加小心, 因为它用在两种不同的方面, 要求两种不同的对待。Plain TEX 为两种情形提供了两个不同的控制系列, `\bmod` 和 `\pmod`: 当'mod'是二元运算时用 `\bmod`(即, 当它出现在两个量之间, 就象加号那样), 当'mod'出现在公式结尾的圆括号中使用 `\pmod`。例如,

```
$\gcd(m,n)=\gcd(n,m\bmod n)$
```
$\gcd(m,n)=\gcd(n,m\bmod n)$

```
$x\equiv y+1\pmod{m^2}$
```
$x\equiv y+1\pmod{m^2}$

'`\bmod`'中的'`b`'表示"binary(二元)"; '`\pmod`'中的'`p`'表示"parenthesized(圆括号)"。注意, `\pmod` 自己插入圆括号; 在圆括号中, 出现在'mod'后面的量将括在大括号中, 只要它不是单个符号。

▶ **练习18.4**
输入‘`$x\equiv0 (\pmod y^n)$`’得到什么结果?

▶ **练习18.5**
看看怎样得到 $\binom{n}{k} \equiv \binom{\lfloor n/p\rfloor}{\lfloor k/p\rfloor}\binom{n \bmod p}{k \bmod p} \pmod{p}$.

要在公式中得到其它字体也是同样的原理。例如, `\bf` 得到的是加粗字体:

`$\bf a+b=\Phi_m$` $\mathbf{a+b=\Phi_m}$

注意, 本例中的整个公式并未都加粗; ‘+’ 和 ‘=’保持不变。Plain TEX 设定象 `\rm` 和 `\bf` 这样的命令只对大写字母 `A` 到 `Z`, 小写字母 `a` 到 `z`, 数字 `0` 到 `9`,大写希腊字母 `\Gamma` 到 `\Omega`, 和象 `\hat` 和 `\tilde` 这样的数学重音起作用。顺便说一下, 在本例中没有使用大括号, 因为 `$` 有编组的作用; `\bf` 改变了当前字体, 但是这个改变是局域的, 因此它不影响公式外面的当前字体。

在 plain TEX 中可用的 bold 字体是“bold roman”, 而不是“bold italic”, 因为后者很少用到。但是, 如果需要, 可以设置 TEX 以使用数学 bold italic 字体(见练习17.20)。数学字体的更大集合还包括手写体, Fraktur 和“blackboard bold”字体; plain TEX 没有这些, 但是其它格式如 $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX 有。

除了 `\rm` 和 `\bf`, 在公式中可以输入 `\cal` 以得到花体大写字母。例如,‘`$\cal A$`’ 得到的是‘$\mathcal{A}$’, ‘`$\cal Z$`’得到的是‘$\mathcal{Z}$’。但是要注意: 它只有大写字母 `A` 到 `Z`; 如果把 `\cal` 用到小写或希腊字母上将得到奇怪的结果。

还有一种表示“数学 italic”的 `\mit` 字体。它对大写希腊字母有作用, 这样得到的是 ($\mathit{\Gamma}, \mathit{\Delta}, \mathit{\Theta}, \mathit{\Lambda}, \mathit{\Xi}, \mathit{\Pi}, \mathit{\Sigma}, \mathit{\Upsilon}, \mathit{\Phi}, \mathit{\Psi}, \mathit{\Omega}$) 而不是 ($\Gamma, \ldots, \Omega$)。当 `\mit` 在起作用时, 普通字母 `A` 到 `Z` 和 `a` 到 `z` 不改变; 它们象通常那样还是 italic 字体, 因为它们一般就是来自数学 italic 字体。反过来, 大写希腊字母和数学重音不受 `\rm` 的影响, 因为它们通常来自 roman 字体。当选择 `\mit` 族时, 数学重音不应该使用, 因为数学 italic 不包含重音符号。

▶ **练习18.6**
利用尽可能少的字输入公式 $\bar{\mathbf{x}}^{\rm T}\mathbf{Mx}=0 \iff \mathbf{x}=\mathbf{0}$。(第一个‘0’是 roman 字体, 第二个是 bold 字体。上标‘T’是 roman 字体。)

▶ **练习18.7**
看看怎样输入‘$S \subseteq \mathit{\Sigma} \iff S \in \mathcal{S}$’。

Plain TEX 还允许在数学公式中使用 italic, slanted 或 typewriter 字母, 只要输入 `\it`, `\sl` 或 `\tt` 即可。但是这些字体只能用在文本尺寸, 因此不要把它们当上标使用。

只要你留心一下, 就想知道为什么要提供 `\mit` 和 `\it`; 原因是 `\mit` 是“数学 italic”(一般用于公式好), 而 `\it` 是“文本 italic”(一般用于文本好)。

`$This\ is\ math\ italic.$` $This\ is\ math\ italic.$

`{\it This is text italic.}` *This is text italic.*

数学 italic 字母要宽一些, 并且间距不同; 它用于大部分公式很好, 但是在数学模式下输入象'$different$'('`$different$`')这样的单词时却不行。在公式中需要宽的'*f*', 但是在文本中不需要。这样的情况几乎从未出现在经典数学中, 但是却常常出现在排版计算机程序上, 因为程序员常常使用多字母的"标识符(identifier)":

```
$\it last:=first$              last := first
$\it x\_coord(point\_2)$       x_coord(point_2)
```

这些例子的第一个表明, 当在数学公式中出现文本 italic 时, TEX 找到了组合字'fi'; 另一个例子表明, 所用的短下划线把标识符名字分开了。当作者排版本手册时, 用'`$\it SS$`'来得到字体名称 *SS*, 因为'`$SS$`'使 S 分得太开了: SS。


▶ 练习18.8
怎样用 plain TEX 命令得到下列列表公式?

$$\mathit{available}+\sum_{i=1}^n\max\bigl(\mathit{full}(i),\mathit{reserved}(i)\bigr)=\mathit{capacity}.$$


▶ 练习18.9
怎样利用 plain TEX 宏排版下列计算机程序?

for $j:=2$ **step** 1 **until** n **do**
  begin $\mathit{accum}:=A[j];\ k:=j-1;\ A[0]:=\mathit{accum};$
  while $A[k]>\mathit{accum}$ **do**
    begin $A[k+1]:=A[k];\ k:=k-1;$
    end;
  $A[k+1]:=\mathit{accum};$
  end.

3. 公式之间的间距 列表公式常常不止一个公式; 例如, 方程经常伴有一个边条件:

$$F_n=F_{n-1}+F_{n-2},\qquad n\ge 2.$$

在这种情况下, 需要告诉 TEX 在逗号后面放多大间距, 因为用 TEX 正常的间距约定就挤在一起了; 如果没有特殊措施, 得到的将是

$$F_n=F_{n-1}+F_{n-2},n\ge 2.$$

传统的排版技术对处理这种情况有一套成熟的标准, 它是基于排版工人称为一个"quad"的间距。因为这些标准看起来在实践中用得很好, 所以 TEX 把它这个习惯为你保留下来了: 当用 plain TEX 格式输入'`\quad`'时, 就在水平方向上得到排版工人的一 quad 间距。类似地, '`\qquad`'给出两个 quad(就象两次那样); 这就是上面 F_n 例子中的正常间距。因此, 建议输入

```
$$ F_n = F_{n-1} + F_{n-2}, \qquad  n \ge 2. $$
```

可能要重申的是, TEX 忽略掉所有数学模式下的空格(当然, 除了'\qquad'后面的空格, 需要用它区分开'\qquad n'和'\qquadn'); 因此, 如果把除它以外的所有空格都去掉, 也可得到同样的结果:

```
$$F_n=F_{n-1}+F_{n-2},\qquad n\ge2.$$
```

只要你需要不同于正常约定的间距, 就必须用诸如 \quad 和 \qquad 的控制系列明确给出它。

过去一 quad 为一个空的方形字, 1 em 宽且 1 em 高——近似为大写字母 M 的尺寸, 就象在第十章中讨论的那样。这个传统没有保留下来: 在 plain TEX 中, 控制系列 \quad 就是'\hskip 1em\relax', 因此 TEX 的 \quad 只有宽度, 没有高度。

也可以象在公式中那样在文本中使用 \quad; 例如, 第十四章演示了怎样把 \quad 用在诗歌上。当 \quad 出现在公式中时, 它表示当前文本字体的 1 em, 与当前数学字体, 尺寸和族无关。因此, 例如, 在下标中的 \quad 与公式主体中是一样宽。

有时候粗心的作者可能把文本段落中的两个公式紧挨着放置。例如, 你可以找到象这样的句子:

The Fibonacci numbers satisfy $F_n=F_{n-1}+F_{n-2},\ \ n\ge2$.

受过正确数学训练的人都知道, 公式应当用单词隔开, 而不仅仅是逗号; 此句的作者至少应该用'for $n\ge2$', 而不只是'$n\ge2$'。但是, 唉, 这样的失误太常见了, 并且许多杰出的数学家都无望地陷在公式堆了。如果不让我们去改变他们的写作样式, 那么至少应该在忽略要插入的相应单词后要插入额外的间距。在这种情况下, 额外的词间间距一般效果很好; 例如, 上面的句子就排版为:

```
... $F_n=F_{n-1}+F_{n-2}$, \ $n\ge2$.}$$
```

这里的'\␣'在视觉上部分弥补了那个不好的样式。

▶ **练习18.10**

把下列段落输入为 TEX 格式, 注意标点和间距; 为了防止不好的断行, 还要用到带子。

Let H be a Hilbert space, C a closed bounded convex subset of H, T a nonexpansive self map of C. Suppose that as $n\to\infty$, $a_{n,k}\to0$ for each k, and $\gamma_n=\sum_{k=0}^\infty(a_{n,k+1}-a_{n,k})^+\to0$. Then for each x in C, $A_nx=\sum_{k=0}^\infty a_{n,k}T^kx$ converges weakly to a fixed point of T.

4. *式子中的间距* 第十六章说过, TEX 自动调整数学公式的间距, 使得它们看起来不错, 而且这一般都可以了。但是有时候你必须帮一下 TEX。数学公式的可能数目太大, 而且 TEX 的间距规则又相当简单, 因此自然会出现例外。当然, 为此希望有间距的精细单位, 而不是从 \␣, \quad 和 \qquad 得到的大块间距。

TEX 放在公式中的间距基本单元叫做*细间距*, *中间距*, 和*厚间距*。为了对这些间距有些体会, 让我们再讨论 F_n 的例子: 厚间距只出现在 = 号前后, 以及 $\ge$ 前后; 中间距出现在 + 前后。细间距略小, 但是也能看出来; 在'$\log\!\log$'和'$\log\log$'之间的差就是细间距。段落中单词之间的正常间距近似等于两个细间距。

TEX 自动把细间距, 中间距和厚间距插入到公式中, 但是只要想添加, 你就可以加上自己要的间距, 使用的控制系列为

\, 细间距(正常为 1/6 个 quad);

`\>` 中间距(正常为 2/9 个 quad);
`\;` 厚间距(正常为 5/18 个 quad);
`\!` 负细间距(正常为 $-1/6$ 个 quad)。

在大多数情况下, 在输入文稿时可以依靠 TEX 的间距, 只有在很少的情况下, 你才发现需要用这四个控制系列插入或去掉间距。

刚才我们看到, `\quad` 间距不随公式的字体而改变, 与所用的字体族也无关。但是细间距, 中间距和厚间距的确随公式字体的大小而变大变小; 这是因为它们的定义用的是 ⟨muglue⟩, 这是用于数学间距的一种特殊粘连。你可以把它看成普通的粘连, 只是它的单位是'`mu`'(数学单位)而不是 `pt`, `cm` 或其它的。例如, 附录 B 包含定义:

```
\thinmuskip = 3mu
\medmuskip = 4mu plus 2mu minus 4mu
\thickmuskip = 5mu plus 5mu
```

它定义了 TEX 要插入公式的细, 中和厚间距。按照这些规定, plain TEX 中的细间距不能伸缩; 中间距可以伸长一点, 当收缩为零; 厚间距可以伸长很多, 但不能收缩。

18mu 等于 1em, 这个 em 来自第 2 族(数学符号族)。换句话说, `\textfont` 2 定义了列表和文字字体中的 `mu` 的 em 值; `\scriptfont` 2 定义了标号字体的 em; `\scriptscriptfont` 2 定义了小标号字体的 em。

你可以把数学粘连插入任何公式, 只要用命令'`\mskip`⟨muglue⟩'即可。例如, '`\mskip 9mu plus 2mu`'就插入当前尺寸的半个 em 的间距, 以及一些伸长度。附录 B 把'`\,`'定义为'`\mskip \thinmuskip`'。类似地, 当无伸缩性时, 可以使用命令'`\mkern`'; '`\mkern18mu`'给出当前尺寸的 1 em 的间距。TEX 坚持 `\mskip` 和 `\mkern` 只用 `mu`; 而 `\hskip` 和 `\kern` (它们也允许在公式中使用)只用不是 `mu` 的单位。

在包含微积分的公式中, 只要把额外的细间距添加在 dx, dy 或 d 前面, 效果就很好; 但是 TEX 不能自动实现它。因此, 一个训练有素的排版者会象下面的例子那样插入'`\,`':

输入	dv
`$\int_0^\infty f(x)\,dx$`	$\int_0^\infty f(x)\,dx$
`$y\,dx-x\,dy$`	$y\,dx-x\,dy$
`$dx\,dy=r\,dr\,d\theta$`	$dx\,dy=r\,dr\,d\theta$
`$x\,dy/dx$`	$x\,dy/dx$

注意, 在最后一个例子中, '`/`'后面不要加'`\,`'。类似地, 如果象

`$$\int_1^x{dt\over t}$$` $$\int_1^x{dt\over t}$$

这样, 也不需要加'`\,`', 因为 dt 单独出现在分数的分子上; 加上会把分数与公式的其它部分在视觉上分开。

▶ **练习18.11**

看看怎样得到列表公式

$$\int_0^\infty \frac{t-ib}{t^2+b^2}e^{iat}\,dt=e^{ab}E_1(ab), \qquad a,b>0.$$



当物理单位出现在公式中时, 应该使用 roman 字体, 并把它同前面的内容用细间距隔开:

`$55\rm\,mi/hr$`	$55\,\rm mi/hr$
`$g=9.8\rm\,m/sec^2$`	$g=9.8\,\rm m/sec^2$
`$\rm1\,ml=1.000028\,cc$`	$\rm 1\,ml=1.000028\,cc$



▶ **练习18.12**

排版下列列表公式, 假定'$\hbar$'由'`\hbar`'得到:

$$\hbar=1.0545\times10^{-27}\,\rm erg\,sec.$$



如果惊叹号(在公式中它表示阶乘)后面跟的是字母, 数字或开分界符, 那么它后面也要插入细间距:

`$(2n)!/\bigl(n!\,(n+1)!\bigr)$`	$(2n)!/\bigl(n!\,(n+1)!\bigr)$
`$${52!\over13!\,13!\,26!}$$`	$${52!\over13!\,13!\,26!}$$

除了这些情形外, 可能偶尔会碰见符号太挤的公式, 或者由于某些不幸的形状组合导致空白太多。在你看到输入得到的结果前是不可能预先知道这些问题的; 看到结果后, 根据你的判断进行最后的修饰, 就得到相当清楚而漂亮的结果了。高超地使用'`\,`'或'`\!`'就可以得到肥瘦相宜的公式, 读者就不会因公式的数学意思而烦心了。根号和多重积分号都需要精细调节。这里给出一些这些情形的例子:

`$\sqrt2\,x$`	$\sqrt2\,x$
`$\sqrt{\,\log x}$`	$\sqrt{\,\log x}$
`$O\bigl(1/\sqrt n\,\bigr)$`	$O\bigl(1/\sqrt n\,\bigr)$
`$[\,0,1)$`	$[\,0,1)$
`$\log n\,(\log\log n)^2$`	$\log n\,(\log\log n)^2$
`$x^2\!/2$`	$x^2\!/2$
`$n/\!\log n$`	$n/\!\log n$
`$\Gamma_{\!2}+\Delta^{\!2}$`	$\Gamma_{\!2}+\Delta^{\!2}$
`$R_i{}^j{}_{\!kl}$`	$R_i{}^j{}_{\!kl}$
`$\int_0^x\!\int_0^y dF(u,v)$`	$\int_0^x\!\int_0^y dF(u,v)$
`$$\int\!\!\!\int_D dx\,dy$$`	$$\int\!\!\!\int_D dx\,dy$$

在每个公式中, 去掉 `\,` 或 `\!` 得到的结果都有些不满意。

这些需要细间距修正的例子的大多数都是由于凑巧而出现的。例如, `$x^2/2$` 中上标在斜线前留下一个洞($x^2/2$); 负细间距就帮着把它填补了。在 `$\sqrt{\,\log x}$` 中, 正的细间距补偿的是'$\log x$'以一个高而直的字母开头这个问题; 等等。但是有两个包含修正的例子是必需的, 因为 TEX 实在照顾不到数学中这么多东西: (1). 在公式 `$\log n(\log\log n)^2$` 中, TEX 不在左圆括号前面插入细间距, 因为它类似于公式 `$\log n(x)$`, 在这里却不需要这样的间距。(2). 在公式 `$n/\log n$` 中, TEX 自动在 `\log` 前插入了不想要的细间距, 因为斜线被看作普通符号, 而且在普通符号和象 `\log` 这样的算符之间一般要有一个细间距。

实际上, 公式中 TEX 的间距规则相当简单。公式如第十七章结尾描述的那样转换成数学列, 而且数学列主要由八种基本"原子"组成: Ord(普通), Op(巨算符), Bin(二元运算), Rel(关系符号), Open(开符号), Close(闭符号), Punct(标点), 以及 Inner(子公式分界符)。出现于命令, 如 `\overline`, `\mathaccent` 或 `\vcenter` 等等中的其它类型的原子都看作 Ord 类型; 分数看作 Inner 类型。下列表用于确定相邻原子间的间距:

左侧原子 \ 右侧原子	Ord	Op	Bin	Rel	Open	Close	Punct	Inner
Ord	0	1	(2)	(3)	0	0	0	(1)
Op	1	1	*	(3)	0	0	0	(1)
Bin	(2)	(2)	*	*	(2)	*	*	(2)
Rel	(3)	(3)	*	0	(3)	0	0	(3)
Open	0	0	*	0	0	0	0	0
Close	0	1	(2)	(3)	0	0	0	(1)
Punct	(1)	(1)	*	(1)	(1)	(1)	(1)	(1)
Inner	(1)	1	(2)	(3)	(1)	0	(1)	(1)

这里的 0, 1, 2, 3 分别表示没有间距, 细间距, 中间距, 厚间距; 用圆括号括起来的表格单元表示只在列表和文本字体中插入, 在标号和小标号时不插入。例如, 在 Rel 行和列是许多单元都是'(3)'; 它表示厚间距正常情况下插入到象'='这样的关系符号前后, 但在下标中不插入。表中的有些单元是'*'; 这是因为 Bin 原子必须放在与二元算符相适应的原子前后, '*'表示 Bin 原子从来不会出现在那种情况下。附录 G 讨论了数学列转换为水平列的准确细节; 只要 TEX 要离开数学模式, 这种转换就开始, 并且原子之间的间距也是那时插入的。

例如, 列表公式

```
$$x+y=\max\{x,y\}+\min\{x,y\}$$
```

将转换为原子系列

x $+$ y $=$ $\max$ $\{$ x $,$ y $\}$ $+$ $\min$ $\{$ x $,$ y $\}$

它们分别是类型 Ord, Bin, Ord, Rel, Op, Open, Ord, Punct, Ord, Close, Bin, Op, Open, Ord, Punct, Ord 和 Close。按照上表插入间距将得到

Ord \> Bin \> Ord \; Rel \; Op Open Ord Punct \, Ord Close \>

$$\text{Bin} \backslash\!> \text{Op Open Ord Punct} \backslash, \text{Ord Close}$$

并且所得到的公式为

$$x \boxplus y = \max\{x, y\} \boxplus \min\{x, y\}$$

即,

$$x + y = \max\{x, y\} + \min\{x, y\} \quad .$$

这个例子没有包括上下标; 但是上下标仅仅是添加原子数而不增加原子的种类。

▶ **练习18.13**
利用表格了给出 TEX 在公式‘`$f(x,y)<x^2+y^2$`’的原子之间中插入什么间距?

Plain TEX 的宏 `\bigl`, `\bigr`, `\bigm` 和 `\big` 得到的都是同样的分界符; 它们之间的唯一差别就是得到不同的间距, 因为它们把分界符变成不同类型的原子: `\bigl` 得到的是 Open 原子, `\bigr` 是 Close, `\bigm` 是 Rel, `\big` 是 Ord。另一方面, 当子公式出现在 `\left` 和 `\right` 之间时, 就把它排版出来, 看作 Inner 原子。因此, 由 `\left` 和 `\right` 封装起来的子公式外面的间距可能比由 `\bigl` 和 `\bigr` 封装起来的多。例如, Ord 后面跟 Inner (来自 `\left`)就得到一个细间距, 但是 Ord 后面跟 Open (来自 `\bigl`)却没有。第十七章的规则意味着, 任意公式中构造的‘`\mathinner{\bigl(`{⟨subformula⟩}`\bigr)}`’正好等价于‘`\left(`⟨subformula⟩`\right)`’, 只要 ⟨subformula⟩ 的结尾不是 Punct, 当然要除了那些不管子公式高度和深度只强制使用 `\big` 分界符的情况。

当公式中有‘`|`’和‘`\|`’时, TEX 的间距规则有时候失效, 因为 $|$ 和 $\|$ 被看作普通符号而不是分界符。例如, 看看公式

`$|-x|=|+x|$`	$\vert - x\vert = \vert + x\vert$
`$\left|-x\right|=\left|+x\right|$`	$\left\vert -x\right\vert = \left\vert +x\right\vert$
`$\lfloor-x\rfloor=-\lceil+x\rceil$`	$\lfloor -x\rfloor = -\lceil +x\rceil$

在第一种情况下间距不对, 因为 TEX 把它看作‘$|$’与‘x’相加了。第二个使用 `\left` 和 `\right` 就对了。第三个表明, 对其它分界符毋需这种修正, 因为 TEX 知道它们是开符号或闭符号。

▶ **练习18.14**
有些数学家把方括号反过来使用表示“开区间”。看看怎样输入下列奇怪的公式? $]-\infty, T[\times\,]-\infty, T[$.

▶ **练习18.15**
研究一下附录 G, 看看公式‘`$x++1$`’所用的间距是什么。哪个加号被看作二元运算?

5. *省略号*(三个小圆点) 如果你仔细地把三个小圆点组合起来输入公式或文本, 数学版面看起来非常好看。虽然在间距固定的打字机上输入‘`...`’看起来不错, 但是当用打印机字体把它输出时得到的结果看起来太挤: ‘`$x...y$`’得到的是‘$x...y$’, 除了上下标外不希望得到这样紧的间距。

省略号可以用两种不同的小圆点——一种比另外一种高——表示出来; 在最好的数学传统中, 这两种是不同的。得到象下面这样的公式一般都是对的:

$x_1+\cdots+x_n$ and $(x_1,\ldots,x_n)$,

但是得到象下面这样的公式却不对:

$x_1+\ldots+x_n$ and $(x_1,\cdots,x_n)$.

附录 B 的 plain TEX 格式让你能很容易解决这个"三个小圆点"的问题, 而且每个人都会羡慕你所得到的优美的公式。办法就是, 当要三个低小圆点($\ldots$)时用 `\ldots`, 用垂直居中的三个小圆点($\cdots$)时用 `\cdots`。

一般地, 最好在 $+$, $-$ 和 $\times$ 号之间, 以及在 $=$, $\leq$, $\subset$ 或其它关系符号之间用 `\cdots`。低的圆点用在逗号之间, 以及无正负号的并列的内容之间。例如,

`$x_1+\cdots+x_n$`	$x_1+\cdots+x_n$
`$x_1=\cdots=x_n=0$`	$x_1=\cdots=x_n=0$
`$A_1\times\cdots\times A_n$`	$A_1\times\cdots\times A_n$
`$f(x_1,\ldots,x_n)$`	$f(x_1,\ldots,x_n)$
`$x_1x_2\ldots x_n$`	$x_1x_2\ldots x_n$
`$(1-x)(1-x^2)\ldots(1-x^n)$`	$(1-x)(1-x^2)\ldots(1-x^n)$
`$n(n-1)\ldots(1)$`	$n(n-1)\ldots(1)$

▶ **练习18.16**

输入公式'$x_1+x_1x_2+\cdots+x_1x_2\ldots x_n$'和'$(x_1,\ldots,x_n)\cdot(y_1,\ldots,y_n)=x_1y_1+\cdots+x_ny_n$'。(提示: 单个升高的圆点叫做'`\cdot`'。)

但是, 在一种重要的特殊情形下, `\ldots` 和 `\cdots` 没有给出正确的间距, 即当它们正好出现在公式结尾时, 或者当它们正好出现在象')'这样的闭分界符前时。在这种情况下, 需要添加额外的细间距。例如, 看看下面的句子:

Prove that $(1-x)^{-1}=1+x+x^2+\cdots\,$.
Clearly $a_i<b_i$ for $i=1$, 2, $\ldots\,$, n.
The coefficients c_0, c_1, $\ldots\,$, c_n are positive.

为了得到第一个句子, 作者输入的是

```
Prove that $(1-x)^{-1}=1+x+x^2+\cdots\,$.
```

没有'`\,`'时, 句点离 `\cdots` 太近。类似地, 第二句输入的是

```
Clearly $a_i<b_i$ for $i=1$,~2, $\ldots\,$,~$n$.
```

注意这里的带子, 它防止第十四章讨论的不好的断行。这样的省略号在数学写作的某些格式中极其普遍, 因此 plain TEX 允许你在段落的文本中只输入‘`\dots`’, 其定义为‘`$\ldots\,$`’。因此第三句可以输入为

```
The coefficients $c_0$,~$c_1$, \dots,~$c_n$ are positive.
```

▶ **练习18.17**
B. C. Dull 想到输入第二个例子的简单办法:

```
Clearly $a_i<b_i$ for~$i=1, 2, \ldots, n$.
```

会出现什么问题?

▶ **练习18.18**
想想看作者怎样输入本书第四章的脚注的?

6. 断行 当段落中出现公式时, TEX 可能不得不把它裂分在两行上。这是注定的命运, 有点象单词的连字化; 除了无路可走外, 我们要尽量避免出现这种情况。

公式只能断行于关系符号后面, 如 $=$, $<$ 或者 $\to$, 或者二元运算后面, 如 $+$, $-$ 或者 $\times$, 其中关系符号或二元运算在公式的“外层”(即没有被封装在 `{...}` 中或者是构造‘`\over`’的一部分)。例如, 如果在段落中输入

```
$f(x,y) = x^2-y^2 = (x+y)(x-y)$,
```

TEX 就可以在 `=` 后面(最好情况)或在 `-`, `+` 或 `-` 后面(不得已情况下)。但是在任何情况下都不会在逗号后面断行——在其后可断行的逗号不应该出现在 `$` 之间。

如果除了在 `=` 之后外不在其它地方断行, 可以输入

```
$f(x,y) = {x^2-y^2} = {(x+y)(x-y)}$
```

因为这些额外的大括号把子公式“冻结”住了, 把它们放在不可断行的盒子中, 盒子的宽度为子公式的自然宽度。但是毋需为此这样做, 因为出现这种情况的机会很少。



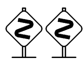
在公式中允许出现“随意乘号”: 如果输入‘`$(x+y)\*(x-y)$`’, 那么 TEX 将把 `\*` 看作象 `\-` 一样; 即, 在此处允许断行, 其 penalty 与连字符一样。但是, 插入的不是连字符, 而是插入文本尺寸的 $\times$。



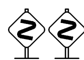
如果的确要允许在外层的某处断行, 可以用 `\allowbreak`。例如, 公式

```
$(x_1,\ldots,x_m,\allowbreak y_1,\ldots,y_n)$
```

出现在段落文本中时, TEX 将允许在‘$(x_1,\ldots,x_m,$’和‘$y_1,\ldots,y_n)$’之间断行。

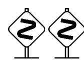


Rel 原子后断行的 penalty 称为 `\relpenalty`, 而 Bin 原子后断行的 penalty 称为 `\binoppenalty`, Plain TEX 的设置为 `\relpenalty=500` 和 `\binoppenalty=700`。在任何特殊情况下, 都可以通过直接在指定的原子后面输入‘`\penalty`⟨number⟩’来改变断行的 penalty; 这样, 你给出的数字就代替了通常的 penalty。例如, 公式‘$x = 0$’通过输入‘`$x=\nobreak0$`’就可以禁止出现断行, 因为 `\nobreak` 的定义是‘`\penalty10000 `’。



▶ **练习18.19**

'`$x=\nobreak0$`'和'`${x=0}$`'得到的结果相同吗?



▶ **练习18.20**

怎样只改动 plain TEX 的宏就可以禁止公式中的所有断行?

7. 大括号 现在已经出现了各种不同的符号, 其中就有符号'{'和'}'; plain TEX 中有几个控制系列可以帮你来处理包括这样符号的公式。

在简单情况下, 大括号用来表示对象的集合; 例如, '$\{a,b,c\}$'表示三个对象 a, b 和 c 的集合。排版这些公式没什么特别的, 只是要记住用 `\{` 和 `\}` 来得到大括号:

`$\{a,b,c\}$`	$\{a,b,c\}$
`$\{1,2,\ldots,n\}$`	$\{1,2,\ldots,n\}$
`$\{\rm red,white,blue\}$`	$\{\rm red,white,blue\}$

在稍复杂的情形下, 集合用一个符合给定条件的一般元素的集合给出; 例如, '$\{\,x\mid x>5\,\}$'表示 x 大于 5 的所有对象的集合。在这种情况下, 应该用控制系列 `\mid` 得到垂直短线, 并且细间距应该插入到大括号中间去:

`$\{\,x\mid x>5\,\}$`	$\{\,x\mid x>5\,\}$
`$\{\,x:x>5\,\}$`	$\{\,x:x>5\,\}$

(有些作者喜欢用冒号而不是'|', 就象第二个例子那样。) 当分界符变大时, 就象在

$$\bigl\{\,\bigl(x,f(x)\bigr)\bigm|x\in D\,\bigr\}$$

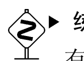
应该用 `\bigk`, `\bigm` 和 `\bigr`; 例如, 刚刚给出的公式可以输入为

```
\bigl\{\,\bigl(x,f(x)\bigr)\bigm|x\in D\,\bigr\}
```

而包含更大分界符要用到 `\Big` 或者 `\bigg` 甚至 `\Bigg`, 就象第十七章讨论的那样。

▶ **练习18.21**

怎样排版公式 $\{\,x^3\mid h(x)\in\{-1,0,+1\}\,\}$?



▶ **练习18.22**

有时候定义集合的条件是相当长的英文文字, 而不是公式; 例如, '$\{\,p\mid p\hbox{ and }p+2\hbox{ are prime}\,\}$'。用一个 hbox 就可得到它:

```
$\{\,p\mid\hbox{$p$ and $p+2$ are prime}\,\}$
```

但是象这样长的公式在段落中会出现问题, 因为 hbox 在段落中不断行, 而且 `\hbox` 中的粘连不随外面行中单词间距的改变而改变。看看怎样得到允许断行的这样的公式。(提示: 在数学模式和水平模式之间来回变换。)

列表公式常常包含另外一种大括号, 来表明不同情形, 就象下式一样:

$$|x|=\begin{cases}x, & \text{if } x\ge 0;\\ -x, & \text{otherwise.}\end{cases}$$

可以用控制系列 \cases 来输入它:

```
$$|x|=\cases{x,&if $x\ge0$;\cr
            -x,&otherwise.\cr}$$
```

仔细观察本例就会发现它用到字符 &, 在第七章我们说过, 它是作为特殊字符保留下来的。在本手册这里是第一次出现特殊字符 & 的例子: 每种情形分为两个部分, 把它们用 & 隔开。在 & 左边是数学公式, 它已经暗中被封装在 $...$ 中了; 在 & 右边是普通文本, 它*没有*放在 $...$ 中。例如, 第二行的'-x,'按数学模式排版, 但是'otherwise'按水平模式排版。& 后的空格被忽略掉。可以有任意多种情形, 但是通常至少有两个。每种情形以 \cr 结束。注意, \cases 构造自己含有'{'; 但是没有相应的'}'。

▶ **练习18.23**
排版列表公式 $f(x)=\begin{cases}1/3 & \text{if } 0\le x\le 1;\\ 2/3 & \text{if } 3\le x\le 4;\\ 0 & \text{elsewhere.}\end{cases}$

你可以在 \cases 中的任意 \cr 后紧跟着插入'\noalign{⟨vertical mode material⟩}', 就象在第二十二章讨论的那样, 因为 \cases 是那章中讨论的一般对齐构造的一个应用。例如, 命令'\noalign{\vskip2pt}'可以在两个情形之间插入小的额外间距。

如果用控制系列 \overbrace 或 \underbrace, 就可以把水平大括号放在部分列表公式的上面或下面。这样的构造象 \sum 这样的巨算符一样, 因此其上下标要变成上下限, 见下面的例子:

```
$$\overbrace{x+\cdots+x}^{k\rm\;times}$$
```

$$\overbrace{x+\cdots+x}^{k\rm\;times}$$

```
$$\underbrace{x+y+z}_{>\,0}.$$
```

$$\underbrace{x+y+z}_{>\,0}.$$

8. *矩阵* 现在渐入佳境了。训练有素的数学家喜欢把公式按行列排列成矩形; 这样的排列称为*矩阵*。Plain TeX 提供了一个叫 \matrix 的控制系列, 用它很任意处理大多数类型的矩阵。

例如, 假定要得到列表公式

$$A=\begin{pmatrix}x-\lambda & 1 & 0\\ 0 & x-\lambda & 1\\ 0 & 0 & x-\lambda\end{pmatrix}.$$

你需要输入的是

```
$$A=\left(\matrix{x-\lambda&1&0\cr
                  0&x-\lambda&1\cr
                  0&0&x-\lambda\cr}\right).$$
```

它非常类似于我们前面看到的 \cases 的构造; 矩阵的每行以 \cr 结尾, 并且用符号'&'分开同一行中不同的单元。但是要注意, 你可以在矩阵的两边放上所需要的 \left 和 \right 分界符; \matrix 在这点上与

`\cases` 不同, 因为 `\cases` 只自动插入一个大'{'。原因是, `\cases` 总是只包含一个左大括号, 而在不同的矩阵构造中要用不同的分界符。另一方面, 最常用的是圆括号, 因此要得到圆括号封装的矩阵, 可以用 `\pmatrix`; 这样上面的例子可以简化为

```
$$A=\pmatrix{x-\lambda&...&x-\lambda\cr}.$$
```



▶ **练习18.24**
用 `\lgroup` 和 `\rgroup` 输入两边公式 $\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \begin{pmatrix} u & x \\ v & y \\ w & z \end{pmatrix}$,



矩阵的各单元一般在列中是居中的。每列的宽度要足以容纳下它所包含的单元, 并且在列之间有一个 quad 的间距。如果要把列中的内容居右, 就要在内容前面加上 `\hfill`; 如果要把列中的内容居左, 就在内容后面加上 `\hfill`。



矩阵的每个单元都与其它单元无关, 它用文本字体排版成数学公式。因此, 例如, 如果在一个单元中使用 `\rm`, 那么它不会影响其它单元。不要使用'`{\rm x&y}`'。

矩阵常常出现通用型的形式, 其中用省略号(即三个小圆点)表示未出现的行或列。可以把省略号放在相应的行和/或列了排版这样的矩阵。Plain TEX 提供了 `\vdots` (垂直省略号)和 `\ddots` (对角省略号), 它们与以前给出的 `\ldots` 一起来构造这样的矩阵。例如, 一般矩阵

$$A=\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$$

由下列方法得到:

```
$$A=\pmatrix{a_{11}&a_{12}&\ldots&a_{1n}\cr
             a_{21}&a_{22}&\ldots&a_{2n}\cr
             \vdots&\vdots&\ddots&\vdots\cr
             a_{m1}&a_{m2}&\ldots&a_{mn}\cr}$$
```

▶ **练习18.25**
怎样得到列矢量 $\begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix}$?



有时候矩阵在上边和左边要放行和列的标记。Plain TEX 为此提供了一个特殊的宏, 叫做 `\bordermatrix`。例如, 列表公式

$$M=\begin{array}{c c} & \begin{array}{ccc} C & I & C' \end{array} \\ \begin{array}{c} C \\ I \\ C' \end{array} & \begin{pmatrix} 1 & 0 & 0 \\ b & 1-b & 0 \\ 0 & a & 1-a \end{pmatrix} \end{array}$$

由下列方法得到:

```
$$M=\bordermatrix{&C&I&C'\cr
                C&1&0&0\cr  I&b&1-b&0\cr  C'&0&a&1-a\cr}$$
```

第一行给出上标记, 它出现在圆括号的上方; 第一列给出左标记, 它居左放置, 就在矩阵自身前面。第一行的第一列一般是空的。注意, `\bordermatrix` 自动插入圆括号, 象 `\pmatrix` 一样。

把矩阵放在段落的文本中一般效果不好, 因为它们太大而无法很好地显示出来。但是, 偶尔可以出现象 $\bigl({1\,1\atop0\,1}\bigr)$ 这样的小矩阵, 它还可以用'`$1\,1\choose0\,1$`'得到。类似地, 小矩阵 $\bigl({a\atop l}{b\atop m}{c\atop n}\bigr)$ 可用下列方法输入:

```
$\bigl({a\atop l}{b\atop m}{c\atop n}\bigr)$
```

用宏 `\matrix` 不能得到这种小的排列。

9. 垂直间距 如果要整理好一个常见的公式, 你已经学会怎样把其内容分开或紧凑一些了, 只要用正的或负的细间距即可。但是这样的调整只对水平尺寸有作用; 如果要把某些内容向高或低处移动怎么办呢? 这可是个高级课题。

附录 B 提供了几个宏, 用它们可以欺骗 TEX, 让它认为某些公式比它们实际情况要大或小; 这样的技巧可以用来把公式的某部分向上下左右移动。例如, 我们已经讨论过第十六章中 `\mathstrut` 和第十七章中 `\strut` 的用法; 这些看不见的盒子使得 TEX 把根号和连分数的分子放在与通常不同的地方上。

如果是任意公式中使用'`\phantom{`⟨subformula⟩`}`', TEX 就给出直接输入'`{`⟨subformula⟩`}`'那样多的间距, 但是公式本身却看不见。因此, 例如, '`\phantom{0}2`'在当前字体中得到与'`02`'同样多的间距, 但是只有 2 实际上出现在页面上。如果要留下一个符号的空白, 而且它正好等于 $\sum$ 的尺寸, 但是你有因为某些原因要手工把那个符号放进去, 这时'`\mathop{\phantom\sum}`'留下的空白正好可以。(这里的'`\mathop`'是为了使这个 phantom (幻影)的性质象 `\sum` 一样, 即, 象一个巨算符。)

比 `\phantom` 更有用的是 `\vphantom`, 它得到的看不见的盒子的高度与深度与相应的 `\phantom` 一样, 但是其宽度为零。因此, `\vphantom` 生成一个垂直 strut, 它可以增加公式的有效高度和深度。Plain TEX 把 `\mathstrut` 定义为'`\vphantom(`'。还有一个叫 `\hphantom`, 其宽度等于 `\phantom` 的宽度, 但是高度和深度都是零。

Plain TEX 还提供了'`\smash{`⟨subformula⟩`}`', 这个宏得到的结果与'`{`⟨subformula⟩`}`'一样, 但是高度和深度都是零。利用 `\smash` 和 `\vphantom`, 可以排版出任意子公式, 并且其高度和深度为所要求的值。例如,

```
\mathop{\smash\limsup\vphantom\liminf}
```

得到一个巨算符, 叫做'$\limsup$', 但是其高度和深度是 `\liminf` 的高度和深度(即, 深度为零)。

▶ **练习18.26**
如果要给某些文本下面划线, 要用的是象

```
\def\undertext#1{$\underline{\hbox{#1}}$}
```

这样的宏。 But this doesn't always work right. 给出一个更好的方法。

还可以利用 \raise 和 \lower 来调整公式中盒子的垂直位置。例如, 公式'$2^{\raise1pt\hbox {$\scriptstyle n$}}$'得到的上标 n 比通常要高 1 pt(是 2^n 而不是 2^n)。注意, 在本例中必须有 \scriptstyle, 因为 \hbox 的内容正常情况下是文本字体, 即使 hbox 出现在上标的位置, 而且 \raise 只能用在盒子上。这种调整的方法并不经常使用, 但是在 \root 宏不能把其变量放在适当位置时要用到。例如,

```
\root\raise⟨dimen⟩\hbox{$\scriptscriptstyle⟨argument⟩$}\of...
```

将把变量向上移给定的量。

不用改变子公式的尺寸, 也不用 \raise, 还可以通过控制 TEX 从数学列转变到水平列时的参数来控制垂直间距。这些参数在附录 G 中讨论; 但是在改变它们时要小心, 因为这样的改变是整体的(即不是组内局域的)。至于这样的改变能做什么, 下面是一个例子: 假定你要设计一个化学排版的格式, 并且希望排版许多化学式, 如'$\rm Fe_2^{+2}Cr_2O_4$'。可能你不希望 $\rm Fe_2^{+2}$ 的下标比 $\rm Cr_2$ 的下标低; 而且不想让用户输入下面这样的怪东西:

```
$\rm Fe_2^{+2}Cr_2^{\vphantom{+2}}O_4^{\vphantom{+2}}$
```

才能得到所有下标在同一水平上的公式 $\rm Fe_2^{+2}Cr_2O_4$. 嗯, 你要做的就是设置'\fontdimen16\tensy=2.7pt'和'\fontdimen17\tensy=2.7pt', 此处假定 \tensy 是你的主符号的字体(\textfont2); 它把所有正常的下标降到比基线低 2.7 pt, 这足以放下包含加号的上标了。类似得, 可以通过改变 \fontdimen14\tensy 来调整上标的位置。有几个参数来表示轴线的位置, 一般分数中分子和分母的位置, 上下限下面和上面的间距, 默认的标尺厚度, 等等。附录 B 给出了详细讨论。

10. 数学高手的特殊要素 TEX 还有几个数学模式的原始命令没有提到。如果要设计特殊格式, 偶尔会用到它们。

如果在粘连或 kern 的描述紧前面有'\nonscript', 那么 TEX 不在标号或小标号字体中使用此粘连或 kern。这样, 例如, 控制系列'\nonscript\;'正好得到本章前面出现的数学间距表给出的'(3)'。

只要 TEX 遇见一个 $ 并且要读入文本中的数学公式, 它就首先读入另一个记号列表——已经由命令 \everymath={⟨token list⟩} 预先定义了。(这类似与第十四章中讨论的 \everypar。) 类似地, \everydisplay={⟨token list⟩} 预先定义了 TEX 在遇见开符号 $$ 后, 即在读入列表公式前要读入的记号列表。利用 \everymath 和 \everydisplay, 可以设置应用于所有公式的特殊约定。

11. 总结 在本章我们讨论的公式种类比任何一本数学书都多。如果你老老实实做了迄今为止的练习, 那么就可以满怀信心地处理几乎任何公式了。

但是下面还有几个练习, 可帮助你复习所学知识。下面每个“难题”都用到本章讨论过的一个或多个原理。作者承认要拌你几个跟头。然而, 如果在看答案之前你做出来了, 或者你对陷阱有提防了, 就会发现这些公式对巩固和完善你的知识有很大好处。

▶ **练习18.27**
难题 1: 看看怎样输入短语‘$n^{\rm th}$ root’, 其中‘$n^{\rm th}$’是上标为 roman 字体的数学公式。

▶ **练习18.28**
难题 2: $\mathbf{S}^{-1}\mathbf{TS}=\mathbf{dg}(\omega_1,\ldots,\omega_n)=\mathbf{\Lambda}.$

▶ **练习18.29**
难题 3: $\Pr(\,m=n\mid m+n=3\,).$

▶ **练习18.30**
难题 4: $\sin18^\circ=\frac14(\sqrt5-1).$

▶ **练习18.31**
难题 5: $k=1.38\times10^{-16}\,\mathrm{erg}/{}^\circ\mathrm{K}.$

▶ **练习18.32**
难题 6: $\bar\Phi\subset NL_1^*/N=\bar L_1^*\subseteq\cdots\subseteq NL_n^*/N=\bar L_n^*.$

▶ **练习18.33**
难题 7: $I(\lambda)=\iint_D g(x,y)e^{i\lambda h(x,y)}\,dx\,dy.$

▶ **练习18.34**
难题 8: $\int_0^1\!\cdots\int_0^1 f(x_1,\ldots,x_n)\,dx_1\ldots\,dx_n.$

▶ **练习18.35**
难题 9: 下面是一个列表公式。

$$x_{2m}\equiv\begin{cases}Q(X_m^2-P_2W_m^2)-2S^2&(m\ \text{odd})\\ P_2^2(X_m^2-P_2W_m^2)-2S^2&(m\ \text{even})\end{cases}\pmod N.$$

▶ **练习18.36**
难题 10: 下面是另一个。

$$(1+x_1z+x_1^2z^2+\cdots\,)\ldots(1+x_nz+x_n^2z^2+\cdots\,)=\frac{1}{(1-x_1z)\ldots(1-x_nz)}.$$

▶ **练习18.37**
难题 11: 另一个。

$$\prod_{j\ge0}\biggl(\sum_{k\ge0}a_{jk}z^k\biggr)=\sum_{n\ge0}z^n\,\Biggl(\sum_{\substack{k_0,k_1,\ldots\ge0\\k_0+k_1+\cdots=n}}a_{0k_0}a_{1k_1}\ldots\,\Biggr).$$

练习18.38

难题 12: 以及,

$$\frac{(n_1+n_2+\cdots+n_m)!}{n_1!\,n_2!\ldots n_m!} = \binom{n_1+n_2}{n_2}\binom{n_1+n_2+n_3}{n_3}\ldots\binom{n_1+n_2+\cdots+n_m}{n_m}.$$

练习18.39

难题 13: 还有另一个列表公式。

$$\Pi_R\begin{bmatrix}a_1,a_2,\ldots,a_M\\ b_1,b_2,\ldots,b_N\end{bmatrix} = \prod_{n=0}^{R}\frac{(1-q^{a_1+n})(1-q^{a_2+n})\ldots(1-q^{a_M+n})}{(1-q^{b_1+n})(1-q^{b_2+n})\ldots(1-q^{b_N+n})}.$$

练习18.40

难题 14: 另一个。

$$\sum_{p\text{ prime}} f(p) = \int_{t>1} f(t)\,d\pi(t).$$

练习18.41

难题 15: 又一个。

$$\{\underbrace{\overbrace{a,\ldots,a}^{k\ a\text{'s}},\overbrace{b,\ldots,b}^{l\ b\text{'s}}}_{k+l\text{ elements}}\}.$$

练习18.42

难题 16: 在复合矩阵中的子矩阵之间有一个 `\smallskip`。

$$\begin{pmatrix}\begin{pmatrix}a & b\\ c & d\end{pmatrix} & \begin{pmatrix}e & f\\ g & h\end{pmatrix}\\ 0 & \begin{pmatrix}i & j\\ k & l\end{pmatrix}\end{pmatrix}.$$

练习18.43

难题 17: 下面要把列居左。

$$\det\begin{vmatrix}c_0 & c_1 & c_2 & \ldots & c_n\\ c_1 & c_2 & c_3 & \ldots & c_{n+1}\\ c_2 & c_3 & c_4 & \ldots & c_{n+2}\\ \vdots & \vdots & \vdots & & \vdots\\ c_n & c_{n+1} & c_{n+2} & \ldots & c_{2n}\end{vmatrix} > 0.$$

练习18.44

难题 18: 下面的主要问题是 $\sum$ 上的撇号。

$$\sum_{x\in A}{}' f(x) \stackrel{\text{def}}{=} \sum_{\substack{x\in A\\ x\neq 0}} f(x).$$

▶ **练习18.45**

难题 19: 现在可以输入这个列表公式了。

$$2\uparrow\uparrow k \stackrel{\rm def}{=} 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\Bigr\}{\scriptstyle k}.$$

▶ **练习18.46**

难题 20: 最后, 当你熟悉所有其它例子后, 再做最好的测试。看看怎样用矩阵得到交换图(许多单元都是空的):

$$\begin{array}{ccccccccc}
&&&&&&0&&\\
&&&&&&\Big\downarrow&&\\
0&\longrightarrow&\mathcal{O}_C&\stackrel{\iota}{\longrightarrow}&\mathcal{E}&\stackrel{\rho}{\longrightarrow}&\mathcal{L}&\longrightarrow&0\\
&&\Big\|&&\Big\downarrow\rlap{\scriptstyle\phi}&&\Big\downarrow\rlap{\scriptstyle\psi}&&\\
0&\longrightarrow&\mathcal{O}_C&\longrightarrow&\pi_*\mathcal{O}_D&\stackrel{\delta}{\longrightarrow}&R^1f_*\mathcal{O}_V(-D)&\longrightarrow&0\\
&&&&&&\Big\downarrow\rlap{\scriptstyle\theta_i\otimes\gamma^{-1}}&&\\
&&&&&&R^1f_*\bigl(\mathcal{O}_V(-iM)\bigr)\otimes\gamma^{-1}&&\\
&&&&&&\Big\downarrow&&\\
&&&&&&0&&
\end{array}$$

12. 忠告 名称很多而且仍然在增加, 因此当你不断排版数学文章时, 可能会不断遇到新的难题。做个人笔记可能是个好办法, 把你处理过的不一目了然的公式都记录下来, 并把所输入的和输出结果都写出来。这样, 当一段时间以后遇见同样的问题将可以回头来找解决方法了。

如果你是给自己排版的数学家, 那么将已经知道怎样把大量复杂的公式排版出来, 并且不会需要可能曲解你的意思的中间人。但是, 别过分陶醉于这个新本领; 能用 TEX 排版出公式并不意味着你找到了与读者交流的最好的渠道。有些记号即使在格式很漂亮时也不太合适。

19. 列表公式

现在你已经知道怎样输入数学公式以得到漂亮的排版了; 数学排版方面的知识基本差不多了。但是还有部分内容, 本章将要讨论它。我们已经讨论过怎样排版单个公式; 但是列表公式常常包括一整组不同的公式, 或者一个大公式的不同片段, 并且在把它们正确地对齐时会出现一点问题。幸运的是, 大量列表公式一般可以分为几个简单的类型。

1. **单行列表公式** 在讨论列表公式排版的一般问题前, 我们先简要地复习一下。如果输入'`$$`⟨formula⟩`$$`', 那么 TeX 将把公式用精美的列表字体显示出来, 自己单独占一行, 并且居中。我们还注意到, 在第十八章, 可以同时显示两个短公式, 只要输入'`$$`⟨formula_1⟩`\qquad`⟨formula_2⟩`$$`'即可; 它把两个公式的问题简化为一个公式的问题。你所得到的两个公式用两个 quad 的间距隔开, 而它们整体行居中。

列表方程常常包含普通文本。第十八章讨论了怎样在不离开数学模式的情况下在公式中输入 roman 字体, 但是在列表公式中排版文本的最好方法是把它放在 `\hbox` 中。这样甚至不需要任何数学内容; 为了得到

要显示的文本

可以只输入'`$$\hbox{要显示的文本}$$`'。但是更好的例子在下面:

$$X_n=X_k \qquad\hbox{if and only if}\qquad Y_n=Y_k \quad\hbox{and}\quad Z_n=Z_k.$$

在这种情况下, 公式和文本都被组合起来, 只要输入

```
$$X_n=X_k \qquad\hbox{if and only if}\qquad
    Y_n=Y_k \quad\hbox{and}\quad Z_n=Z_k.$$
```

注意, `\qquad` 出现在'if and only if'两边, 但是在'and'两边只有一个 `\quad`; 这清楚地表明了列表公式的 Y 和 Z 的关系比它们与 X 的关系更近。

现在看看列表公式

$$Y_n=X_n\bmod p \quad\hbox{and}\quad Z_n=X_n\bmod q \qquad\hbox{for all }n\ge0.$$

你知道怎样输入吗? 一种方法是

```
$$Y_n=X_n\bmod p \quad\hbox{and}\quad Z_n=X_n\bmod q
    \qquad\hbox{for all }n\ge0.$$
```

注意, 这里 hbox 中的'`all`'后面有一个空格, 因为当空格出现在公式区时会去掉。但是一旦你理解了 TeX 的模式思想, 就有更好和更合理的方法: 可以输入

```
... \qquad\hbox{for all $n\ge0$.}$$
```

噢——是列表数学模式中的水平模式中的数学模式。但是用这种方法, 文稿正好反映出你要表达的意思, 而前一种方法('`all`'后面加空格)看起来有点不得已。

▶ **练习19.1**

输入下列四个列表公式(一次一个):

$$\sum_{n=0}^\infty a_nz^n \qquad \hbox{converges if} \qquad |z| < \Bigl(\limsup_{n\to\infty}\root n\of{|a_n|}\,\Bigr)^{-1}.$$

$${f(x+\Delta x)-f(x)\over\Delta x}\to f'(x) \qquad \hbox{as } \Delta x\to0.$$

$$\|u_i\|=1, \qquad u_i\cdot u_j=0 \quad \hbox{if } i\ne j.$$

$$\it\hbox{The confluent image of} \quad \left\{\matrix{\hbox{an arc}\hfill\cr\hbox{a circle}\hfill\cr\hbox{a fan}\hfill\cr}\right\} \quad \hbox{is} \quad \left\{\matrix{\hbox{an arc}\hfill\cr\hbox{an arc or a circle}\hfill\cr\hbox{a fan or an arc}\hfill\cr}\right\}.$$

▶ **练习19.2**

当要排版的公式为

$$y={1\over2}x$$

或者类似这样简单的东西时, 有时候列表字体太大了。有一天 User 希望通过输入'`$$y={\scriptstyle1 \over\scriptstyle2}x$$`'来弥补一下, 结果得到的

$$y={\scriptstyle1\over\scriptstyle2}x$$

根本不是他想要的。当不希望在列表公式中出现大的分数时, 直接得到'$y={1\over2}x$'的正确方法是什么?

▶ **练习19.3**

如果'`$$`⟨formula⟩`$$`'和'`$$\hbox{$`⟨formula⟩`$}$$`'所得到的结果之间有什么差别?

▶ **练习19.4**

你可能已经注意到了, 本手册的列表公式不是居中的; 作为书籍设计的一部分, 列表公式一般与段落左边的缩进对齐, 因为这不是一般的书。看看怎样排版出下面这样不居中的公式:

$$1-{1\over2}+{1\over3}-{1\over4}+\cdots=\ln 2$$

如果你有以前排版数学文章的经验, 可能会想, "方程编号放在哪里? 本书什么时候讨论它?" 的确, 现在是该讨论在列表公式外边好远才偷偷摸摸出现的小标记了。如果输入

`$$`⟨formula⟩`\eqno`⟨formula⟩`$$`

TeX 就列表显示出第一个公式, 并且还把方程编号(第二个公式)放在右页边处。例如,

```
$$x^2-y^2 = (x+y)(x-y).\eqno(15)$$
```

得到的是:

$$x^2-y^2=(x+y)(x-y). \qquad (15)$$

也可以把方程编号放在左页边处, 用的是 `\leqno`。例如,

```
$$x^2-y^2 = (x+y)(x-y).\leqno(16)$$
```

得到的是:

$$x^2-y^2=(x+y)(x-y). \tag{16}$$

(16) 在左边。

注意, 方程编号总是放在第二位, 即使要出现在左边。从 `\eqno` 到 `\leqno` 命令到结束列表公式的 `$$` 的所有东西都是方程编号。因此, 不允许在同一列表方程中出现两个方程编号; 但是有一种方法可以绕过这个限制, 我们后面要讨论它。

现在, 人们越来越多地使用方程的右边编号, 因为列表公式常常出现在句子或条款的结尾处, 右边约定使得编号与条款不混淆。还有, 当列表公式跟在一个短行后面时, 还常常可以节约空间, 因为在列表公式上面需要的间距更少; 用 `\leqno` 时就不能这样节约空间, 因为没有可以叠起来的空白。例如, 在我们关于 `\eqno` 和 `\leqno` 的示例中, 公式 (15) 上面的空间比公式 (16) 要少, 虽然公式和文本这些其它东西都是一样的。

如果你仔细观察上面的 (15) 和 (16), 就会发现列表公式已经居中了, 而没有计及方程编号的出现。但是当公式很大时, TEX 要确保方程与编号之间互不干扰; 方程编号甚至可以单独放一行。

▶ **练习19.5**
怎样单独下列列表公式?

$$\prod_{k\ge0}\frac{1}{(1-q^kz)}=\sum_{n\ge0}z^n\Bigg/\prod_{1\le k\le n}(1-q^k). \tag{16$'$}$$

▶ **练习19.6**
方程编号是用文本字体排版的数学公式。因此, 怎样才能得到象'(3–1)'(有一个短破折号)这样的公式编号?

▶ **练习19.7**
User 试着输入了'`\eqno(*)`' and '`\eqno(**)`', 他高兴地发现, 得到的方程编号是'$(*)$'和'$(**)$'。[他为没有出现'(*)'和'(**)'而感到困惑。] 但是, 几个月后, 他输入'`\eqno(***)`'却得到意想不到的结果。这个结果是什么?

应该在本手册的某些地方讨论一下 TEX 到底是怎样排版列表公式的: 即, 怎样把它们居中, 怎样放方程编号, 怎样在上下插入额外间距等等。嗯, 现在该讨论它们了。它们是有些复杂, 因为它们与象 `\parshape` 这样的东西互相影响, 并且包括了几个还未讨论的参数。这些规则的目的就是当出现列表公式时, 要把什么样的盒子, 粘连和 penalty 放在当前垂直列中。

如果列表公式出现在段落的第 4 行后面, 在列表公式开始时, TEX 的内部寄存器 `\prevgraf` 就等于 4。列表公式设置的行数为 3, 因此当在列表公式结束处恢复到段落时, `\prevgraf` 就变成 7 (除非在其间你改动过 `\prevgraf`)。在读入开符号 `$$` 后, TEX 紧接着就指定了三个 ⟨dimen⟩ 参数的特殊值: `\displaywidth` 设定的是行的宽度 z, `\displayindent` 按照当前段落的形状和悬挂缩进设定了第 `\prevgraf` $+\,2$ 行的偏移量 s。(一般情况下, `\displaywidth` 与 `\hsize` 是一样的, 而 `\displayindent` 等于零, 但是段落形状可以象第十四章讨论的那样变化。) 还有, `\predisplaysize` 设定了列表公式前面行的有效宽度 p, 如下: 如果前面没有行(比如, 如果 `$$` 前面是 `\noindent` 或者是另一个列表公式的闭符号

`$$`), p 就设定为 $-16383.99999\,\mathrm{pt}$(即, 可用的最小尺寸, $-$`\maxdimen`)。否则, TeX 要根据前面的行所形成的 hbox 把 p 设定为此 hbox 中最右边盒子的右边界的位置, 加上把封装盒子向右移的缩进, 再加上当前字体的 2 em。但是, 如果 p 的这个值由此盒子中粘连的伸缩所确定——例如, 如果 `\parfillskip` 粘连是有限的, 使得在其前面的内容不是其自然宽度——那么 p 就设定为 `\maxdimen`。(这不是常常出现, 但是它保持了 TeX 系统的独立性, 因为 p 不依赖于不同计算机的四舍五入得到的量。) 注意, `\displaywidth` 和 `\displayindent` 不受 `\leftskip` 和 `\rightskip` 影响, 但是 `\predisplaysize` 却受其影响。在列表公式读入后, TeX 才用到 `\displaywidth`, `\displayindent` 和 `\predisplaysize` 的值, 其讨论见下面; 如果要得到不同的排版, 可用检验和/或改变它们。

在读入列表公式后, TeX 把它从数学列转换到列表字体的水平列 h, 就象在附录 G 中讨论的那样。如果出现方程编号, 就把它按文本字体处理, 并且放在宽度为自然宽度的 hbox 中。现在进行烦琐的处理: 设 z, s 和 p 是 `\displaywidth`, `\displayindent` 和 `\predisplaysize` 的当前值。如果没有方程编号, 就设 q 和 e 为零; 否则, 设 e 为方程编号的宽度, 并且设 q 等于 e 加上符号字体(即, `\textfont2`)的一个 quad。设 w_0 是列表公式 h 的自然宽度。如果 $w_0+q\le z$, 就把列 h 包装进宽度为自然宽度 w_0 的 hbox b 中。但是如果 $w_0+q>z$ (即, 如果列表公式太宽按照自然宽度放不下), 那么 TeX 执行下列"挤压程序": 如果 $e\ne 0$ 并且在列表公式 h 中有足够的收缩性把其宽度减小到 $z-q$, 那么列 h 就包装进宽度为 $z-q$ 的 hbox b 中。否则, 把 e 设为零, 并且把列 h 包装进宽度为 $\min(w_0,z)$ 的 hbox b 中(可能溢出)。

(续) 现在 TeX 试着把列表公式居中, 而不管方程编号。但是如果这样的居中使得它与此编号太近("太近"就是它们之间的间距小于宽度 e), 那么方程在剩下的间距上居中, 或者放在离方程编号尽可能远的地方。只有当列 h 上的第一个项目是粘连时, 才选择后一种方法, 因为 TeX 假定这样的粘连放在那里是为了精确地控制间距。但是我们要正式给出规则: 设 w 是盒子 b 的宽度。通过首先设 $d=\frac12(z-w)$ 来计算位移量 d——当放置盒子 b 时要用到它。如果 $e>0$ 并且 $d<2e$, 那么 d 被重新设定为 $\frac12(z-w-e)$ 或者零, 其中只有列 h 的开头为一个粘连项目时才选择零。

(续) 现在 TeX 要把这些东西放在当前垂直列, 紧接在前面段落所构造的内容后面。首先出现的是一个 penalty 项目, 它的成本是整数参数 `\predisplaypenalty`。接着出现的是粘连。如果 $d+s\le p$, 或者如果方程编号在左边(`\leqno`), 那么 TeX 分别把 g_a 和 g_b 设置为给定的粘连项目参数 `\abovedisplayskip` 和 `\belowdisplayskip`; 否则 g_a 和 g_b 就变成对应于 `\abovedisplayshortskip` 和 `\belowdisplayshortskip` 的粘连项目。[转换: 如果 predisplaysize 足够小, 使得它与列表公式不重叠, 那么列表公式上下的粘连就比有重叠时的粘连要"小"。] 如果 $e=0$ 并且如果有 `\leqno`, 那么方程编号就就象一个 hbox 一样追加在自己后面, 向右平移 s 并且在前面象通常那样放上行间粘连; 还要追加一个无限大 penalty, 来防止这个编号与列表方程之间断行。否则, 就把粘连项目 g_a 放在垂直列中。

(续) 现在出现的是列表方程自己了。如果 $e\ne 0$, 那么方程编号的盒子 a 就与公式的盒子 b 如下组合起来: 设 k 为宽度为 $z-w-e-d$ 的 kern。在 `\eqno` 的情况下, 盒子 b 由包含 (b,k,a) 的 hbox 代替; 在 `\leqno` 的情况下, 盒子 b 由包含 (a,k,b) 的 hbox 代替, 并且把 d 设为零。在所有情形下, 接下来都把盒子 b 追加到垂直列中, 向右平移 $s+d$。

(续) 最后一个任务就是追加上粘连或跟在列表方程下面的方程编号。如果有一个 `\eqno` 并且 $e=0$, 就在垂直列中放一个无限大 penalty; 后面是方程编号的盒子 a, 它向右平移 $s+z$ 减掉其宽度的距离; 再接下来是成本为 `\postdisplaypenalty` 的值的 penalty 项目。否则, 首先追加的是 `\postdisplaypenalty` 的 penalty 项目, 接着是由上面给出的 g_b 的粘连项目。现在, TEX 把 `\prevgraf` 增加 3 并且返回水平模式, 就回复到段落。

这些规则的一个结果就是通过把方程编号盒子的宽度变成零, 即, 使用'`\eqno\llap{$`⟨formula⟩`$}`'或者'`\leqno\rlap{$`⟨formula⟩`$}`', 你可以把它单独放在一行上。它使得 $e=0$, 并且条件 $e=0$ 在逻辑上控制着 TEX 的放置方法, 就象刚刚讨论的规则那样。

Plain TEX 设置 `\predisplaypenalty=10000`, 因为按照惯例, 好的排版应该避免把列表公式放在页面紧顶部的地方。如果你希望或者不希望正好在列表公式前或后分页, 就可以改变 `\predisplaypenalty` 和 `\postdisplaypenalty`。例如, '`$$\postdisplaypenalty=-10000`⟨formula⟩`$$`' 就强制分页, 把公式放在最底的一行了。这种强制分页的方法比在 `$$...$$` 输入 `\eject` 要好; 这样的 eject (它跟在列表公式下面的 `\belowdisplayskip` 粘连后)得到一个短页面, 因为它在底部留下了不想要的粘连。

▶ **练习19.8**
详细阅读上面的规则, 推导出公式

```
$$\quad x=y \hskip10000pt minus 1fil \eqno(5)$$
```

中'$x=y$'的最后位置, 假定没有悬挂缩进。`\leqno` 呢?

TEX 还允许"对齐的列表公式", 它们不在数学模式下处理, 因为在外层没有包含公式。对齐的列表公式由一般形式为

`$$`⟨assignments⟩`\halign{`⟨alignment⟩`}`⟨assignments⟩`$$`

的命令生成, 其中 ⟨assignments⟩ 是象参数那样的可选内容, 它不输出任何数学列。在这样的列表公式中, `\halign` 就象它出现在垂直模式中那样处理, 并且构造出象通常那样的垂直列 v, 只是对齐的每行后向右平移了 `\displayindent`。在对齐和闭对齐都处理完后, TEX 将把一个 `\predisplaypenalty` 项目和一些 `\abovedisplayskip` 粘连放在主垂直列, 接着是 v, 再接下来是一个 `\postdisplaypenalty` 项目和 `\belowdisplayskip` 粘连。这样, 对齐的列表公式本质上就象普通的对齐一样了, 只是它们可以插在段落中间; 还有, 它们就象其它列表公式那样嵌入在粘连和 penalty 中。`\displaywidth` 和 `\predisplaysize` 不会影响结果, 虽然你可以在 `\halign` 使用它们。至于 `\prevgraf`, 这个对齐列表只看作三行长。

2. *多行列表公式* 好, 列表公式非常有用。但是当你要排版许多文稿时, 就会发现有些列表公式只用有无编号的单行方程这种简单式样不合适。Plain TEX 提供了特殊控制系列来解决大多数剩下的问题。

多行列表公式通常由几个方程组成, 它们的'='应当对齐, 就象下面这样:

$$X_1+\cdots+X_p=m,$$
$$Y_1+\cdots+Y_q=n.$$

这样的列表方程建议使用的命令是 `\eqalign`, 它用到特殊标记 `&` 和 `\cr`, 这些我们在第十八章中与 `\cases` 和 `\matrix` 有关的地方已经见过。下面就是上面公式的输入方法:

```
$$\eqalign{X_1+\cdots+X_p&=m,\cr
           Y_1+\cdots+Y_q&=n.\cr}$$
```

在 `\eqalign` 中可以有任意个方程; 一般样式为

`\eqalign{`⟨left-hand side$_1$⟩`&`⟨right-hand side$_1$⟩`\cr`
⟨left-hand side$_2$⟩`&`⟨right-hand side$_2$⟩`\cr`
⋮
⟨left-hand side$_n$⟩`&`⟨right-hand side$_n$⟩`\cr}`

其中每个 ⟨right-hand side⟩ 以要对齐的符号开头。例如, 每个右边都常常以 = 开头。方程用列表字体排版。

▶ **练习19.9**
在实践中, 要对齐的公式的左边常常是空的, 对齐经常出现在其它符号以及 = 上。例如, 下列列表公式就比较典型; 看看你是否知道作者是怎样输入的:

$$\begin{aligned}T(n)\le T(2^{\lceil\lg n\rceil})&\le c(3^{\lceil\lg n\rceil}-2^{\lceil\lg n\rceil})\\&<3c\cdot 3^{\lg n}\\&=3c\,n^{\lg 3}.\end{aligned}$$

`\eqalign` 得到的结果是一个垂直居中的盒子。这使得很容易得到象下面这样的公式:

$$\left\{\begin{aligned}\alpha&=f(z)\\ \beta&=f(z^2)\\ \gamma&=f(z^3)\end{aligned}\right\}\qquad\left\{\begin{aligned}x&=\alpha^2-\beta\\ y&=2\gamma\end{aligned}\right\}.$$

可以直接在同一行中使用 `\eqalign` 两次:

```
$$\left\{
\eqalign{\alpha&=f(z)\cr \beta&=f(z^2)\cr \gamma&=f(z^3)\cr}
\right\}\qquad\left\{
\eqalign{x&=\alpha^2-\beta\cr y&=2\gamma\cr}\right\}.$$
```

▶ **练习19.10**
练练手, 看看怎样得到带编号的两行的列表公式:

$$\begin{aligned}P(x)&=a_0+a_1x+a_2x^2+\cdots+a_nx^n,\\P(-x)&=a_0-a_1x+a_2x^2-\cdots+(-1)^na_nx^n.\end{aligned}\qquad(30)$$

[提示: 利用 `\eqalign` 得到垂直居中的盒子; 方程编号'(30)'假定出现在两行的中间。]

▶ **练习19.11**

如果在 `\eqalign` 的一个方程中你忘记输入 `&` 会出现什么情况?

多行公式有时候用奇怪的方法放在一起, 并且偶尔你希望把行发散或紧凑一些。如果在任意 `\cr` 输入‘`\noalign{\vskip`⟨glue⟩`}`’, 那么 TEX 将在规定的行后面插入给定量的额外粘连。例如,

```
\noalign{\vskip3pt}
```

将把 3 pt 的额外间距放在行之间。用同样的方法也可以改变第一行前面的间距。

当你有几个要对齐的方程和几个方程编号时, 就出现更复杂的情况了。或许有些行有编号, 有些又没有编号:

$$\begin{aligned}(x+y)(x-y)&=x^2-xy+yx-y^2\\&=x^2-y^2; \qquad (4)\\(x+y)^2&=x^2+2xy+y^2. \qquad (5)\end{aligned}$$

为此 plain TEX 提供了 `\eqalignno`; 你可以象 `\eqalign` 那样使用它, 但是在每个有方程编号的行上, 只需要在 `\cr` 前面添加‘`&`⟨equation number⟩’即可。上面的例子由下面得到:

```
$$\eqalignno{(x+y)(x-y)&=x^2-xy+yx-y^2\cr
    &=x^2-y^2;&(4)\cr
  (x+y)^2&=x^2+2xy+y^2.&(5)\cr}$$
```

注意, 如果没有方程编号, 那么第二个 `&` 就可以省略。

还有一个 `\leqalignno`, 它把方程编号放在左边。在这种情况下, 输出结果中‘(4)’就相应地出现在方程前面:

$$\begin{aligned}(4) \qquad (x+y)(x-y)&=x^2-xy+yx-y^2\\&=x^2-y^2;\\(5) \qquad (x+y)^2&=x^2+2xy+y^2.\end{aligned}$$

虽然方程编号出现在左边, 你却仍然要把它们输入在右边, 就象 `\leqno` 那样; 换句话说, 要得到前面的列表公式, 应该输入的是‘`$$\leqalignno{(x+y)(x-y)&...&(4)\cr...}$$`’。

注意: `\eqalignno` 和 `\leqalignno` 都把方程组居中, 而不管方程编号的宽度。如果方程或它们的编号太宽, 可能会重叠, 但是却不给出错误信息。

▶ **练习19.12**

排版下列列表公式:

$$\begin{aligned}(9) \qquad \gcd(u,v)&=\gcd(v,u);\\(10) \qquad \gcd(u,v)&=\gcd(-u,v).\end{aligned}$$

▶ **练习19.13**

试试另一个, 只是为了实用:

$$\eqalignno{\left(\int_{-\infty}^{\infty}e^{-x^2}\,dx\right)^2&=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}e^{-(x^2+y^2)}\,dx\,dy\cr &=\int_0^{2\pi}\int_0^{\infty}e^{-r^2}r\,dr\,d\theta\cr &=\int_0^{2\pi}\left(-{e^{-r^2}\over2}\bigg|_{r=0}^{r=\infty}\right)d\theta\cr &=\pi.&(11)\cr}$$

虽然 \eqalign 和 \eqalignno 看起来几乎一样, 但是它们之间有一个根本的不同: \eqalign 得到的是一个垂直居中的盒子, 宽度不会超过所需要的宽度; 但是 \eqalignno 生成了一组行, 宽度为整个列表公式的宽度(每行都延伸到两个边界)。因此, 例如, 可以在一个列表公式中使用几次 \eqalign, 但是只能用一次 \eqalignno。如果把 \eqno 与 \eqalign 联合起来用, 得到的结果还可以, 但是如果把 \eqno 与 \eqalignno 联合起来用, 就会得到某些奇怪的错误信息。

附录 B 中的定义揭示了 \eqalign 和 \eqalignno 的性质如此不同的原因: \eqalign 的定义是 \vcenter{\halign{...}}, 而 \eqalignno 的定义是 \halign to\displaywidth{...}; 因此, 宏 \eqalignno 得到的是"对齐的列表公式"。

\eqalign 和 \eqalignno 之间的这个差别有两个有意思的后果: (1). 不可能在 \eqalign 中分页, 但是在 \eqalignno 中可以。实际上, 如果在某行的 \cr 后插入'\noalign{\break}', 就可以在此行后强制分页。如果要禁止 \eqalignno 中的所有分页, 可以设置 \interdisplaylinepenalty=10000; 或者可以把整个公式组放在 \vbox 中:

```
$$\vbox{\eqalignno{...}}$$
```

(2). 还可以在两个方程之间插入一行文本, 而不影响对齐结果。例如, 看看下面两个列表公式:

$$x=y+z$$

and

$$x^2=y^2+z^2.$$

它们实际上由一个列表公式输入:

```
$$\eqalignno{x&=y+z\cr
  \noalign{\hbox{and}}
  x^2&=y^2+z^2.\cr}$$
```

因此, 它们的 $=$ 对齐不是因为凑巧。有时候可能要调整所插入文本上下的间距, 只要输入 \vskip 或者两边放上 \noalign{...} 即可。顺便说一句, 整个例子还表明, 可以在无方程编号时使用 \eqalignno。

▶ **练习19.14**

如果最后一个例子中用 `\eqalign` 代替 `\eqalignno` 会出现什么情况?

▶ **练习19.15**

我们的朋友笨笨又遇到麻烦了, 当他要把方程编号从原来的位置上抬高时, 输入的是:

```
$$\eqalignno{...&\raise6pt\hbox{(5)}\cr}$$
```

他的失误在哪里? 怎样才能实现他的愿望?

对其它类型的列表公式, plain TEX 提供了 `\displaylines`, 用它可以用任意想要的方式输入任意个公式, 但是没有对齐。一般形式为

$$\displaylines{⟨displayed formula$_1$⟩\cr
⟨displayed formula$_2$⟩\cr
⋮
⟨displayed formula$_n$⟩\cr}$$

每个公式都要居中, 因为 `\displaylines` 在每行的两边都放上了 `\hfil`; 插入比 `\hfil` 更强的 `\hfill` 可以把它左对齐或右对齐。

▶ **练习19.16**

用 `\displaylines` 来排版下列三行的列表公式:

$$x \equiv x; \qquad (1)$$

$$\text{if}\quad x \equiv y \quad\text{then}\quad y \equiv x; \qquad (2)$$

$$\text{if}\quad x \equiv y \quad\text{and}\quad y \equiv z \quad\text{then}\quad x \equiv z. \qquad (3)$$

如果你仔细观察本章中多行列表公式, 就会发现其基线比正常文本的基线要分开得更宽; 数学出版界这样做是为了使得列表公式更易于阅读。遵循这个惯例, `\eqalign` 等自动增加其 `\baselineskip`。如果你用 TEX 的原始命令 `\halign` 而不是 plain TEX 的宏来制作多行列表公式, 就可能要做同样的基线调整, 这只需输入'`$$\openup1\jot \halign{...}$$`'即可。宏 `\openup` 增加 `\lineskip`, `\lineskiplimit` 以及 `\baselineskip`。如果输入的是'`\openup2\jot`', 基线就多分开 2 个单位, plain TEX 的设置为增加 3 pt 的单位。因为 `$$...$$` 有编组的作用, 所以当列表公式结束时 `\openup` 的影响就消失了。`\openup` 后面可以跟任何 ⟨dimen⟩, 但是习惯上用 `\jot` 符号来表示这个量, 而不用绝对单位; 这样你的文稿就可用于各种格式了。

Plain TEX 的宏 `\displaylines`, `\eqalignno` 和 `\leqalignno` 以'`\openup1\jot`'开头。如果不希望行分开太多, 可以取消它, 比如输入'`$$\openup-1\jot \eqalignno{...}$$`', 因为 `\openup` 有累加性。

假定你要自制一个列表公式, 其一般形式为'`$$\openup1\jot \halign{...}$$`'; 并且为了使用方便, 假定使用的是 plain TEX 的正常约定, 使得 `\jot=3pt` 和 `\baselineskip=12pt`。因此, 宏

\openup 把 baselineskip 的距离变成 15 pt。随后, 列表公式的文本行紧上面的基线将在列表公式最上面基线上面 15 pt, 再加上 \abovedisplayskip 的地方。但是当回到段落时, 其下一个基线只是在列表公式最低基线下 12 pt, 再加上 \belowdisplayskip 的地方, 因为参数 \baselineskip 已经回到其正常值。为了弥补这个差, 在宏 \eqalignno 和 \displaylines 的第一行前面添加上'\noalign{\vskip$-d$}', 其中 d 是多分开的量。

3. 长公式 我们关于数学排版的讨论差不多结束了; 现在只想要再处理一个问题: 当公式太长在一行放不下时怎么办?

例如, 假定你碰到方程

$$\sigma(2^{34}-1,2^{35},1)=-3+(2^{34}-1)/2^{35}+2^{35}/(2^{34}-1)+7/2^{35}(2^{34}-1)-\sigma(2^{35},2^{34}-1,1).$$

那么就不得不把它裂分; TEX 只是尽力通过把 + 和 − 间距收缩到零, 从而把所有内容挤在一起, 但是这样仍得到溢出的行。

我们试着在'+7'前面把方程裂分。一个常用的方法是输入

```
$$\eqalign{\sigma(2^{34}-1,2^{35},1)
&=-3+(2^{34}-1)/2^{35}+2^{35}\!/(2^{34}-1)\cr
&\qquad+7/2^{35}(2^{34}-1)-\sigma(2^{35},2^{34}-1,1).\cr}$$
```

它得到的是 which yields

$$\begin{aligned}\sigma(2^{34}-1,2^{35},1)&=-3+(2^{34}-1)/2^{35}+2^{35}\!/(2^{34}-1)\\&\qquad+7/2^{35}(2^{34}-1)-\sigma(2^{35},2^{34}-1,1).\end{aligned}$$

这个方法是把单行长公式变成两行的公式, 在第二行用 \qquad 给第一行的 = 留下对应的空白。

▸ **练习19.17**

怎样输入下列列表公式?

$$\begin{aligned}x_nu_1+\cdots+x_{n+t-1}u_t&=x_nu_1+(ax_n+c)u_2+\cdots\\&\qquad+\bigl(a^{t-1}x_n+c(a^{t-2}+\cdots+1)\bigr)u_t\\&=(u_1+au_2+\cdots+a^{t-1}u_t)x_n+h(u_1,\ldots,u_t).\qquad(47)\end{aligned}$$

怎样把长的列表公式裂分成几行是一项技术; TEX 从不试图把它们裂分, 因为实在没有一组适当的规则。一般情况下, 数学文稿的作者是最好的裁判, 因为断点的位置依赖于数学含义的微妙因素。例如, 常常希望强调某些对称性或者公式的其它潜在意义, 并且这样的东西要求确实理解了此公式中要表达的意思。

然而, 对处理列表公式中长公式, 可能有几个笨规则, 因为有几个最好的数学排版者要遵循的原理:

a) 虽然段落中的公式总是在二元运算和关系符号之后断行, 但是列表公式总是在它们之前断行。因此, 不要在 $\sigma(\ldots)$ 这个例子的'(2^{34}-1)+'处结束第一行; 应该在'(2^{34}-1)'处结束它, 并且以'+'开始第二行。

b) 当方程在二元运算前被裂分时, 第二行至少在包含第一行此二元运算的最内层子公式右边的两个 quad 后面才开始。例如, 如果要在 ⟨formula₁⟩ 和 ⟨formula₂⟩ 之间的加号处裂分公式

```
$$\sum_{0<k<n}\left(⟨formula1⟩+⟨formula2⟩\right)$$
```

那么几乎是人为地把第二行的加号出现在左侧大圆括号(它由'`\left(`'得到)的右边。

在刚刚讨论的例子中, 想要特别注意要把公式裂分为两行, 因为 `\left` 和 `\right` 分界符不能单独使用; 不能在第一行只用 `\left` 和在第二行只用 `\right`。还有, 如果要得到同样大小的两个分界符——即使它们在不同的行, 那么最好的方法通常是自己选定分界符的尺寸; 例如, 如果 `\bigg` 最适合, 可以输入

```
$$\eqalign{\sum_{0<k<n}\biggl(&⟨formula1⟩\cr
            &\qquad{}+⟨formula2⟩\biggr)\cr}$$
```

注意, 在本例中, 标记 & 后面没有 = 号, 它们仅仅是标记要对齐的点。

还有另一种方法了裂分长公式, 有时候把它称为双行形式。其想法是, 公式的第一部分几乎居左, 第二部分几乎居右, 其中"几乎居···"表示"差一个 quad"。这样, 前面讨论的长方程 $\sigma(\ldots)$ 的双行形式为

$$\sigma(2^{34}-1,2^{35},1)=-3+(2^{34}-1)/2^{35}+2^{35}/(2^{34}-1)$$
$$+7/2^{35}(2^{34}-1)-\sigma(2^{35},2^{34}-1,1).$$

用 `\displaylines` 不难得到这种双行的效果:

```
$$\displaylines{\quad\sigma(2^{34}-1,2^{35},1)
  =-3+(2^{34}-1)/2^{35}+2^{35}\!/(2^{34}-1)\hfill\cr
\hfill{}+7/2^{35}(2^{34}-1)-\sigma(2^{35},2^{34}-1,1).\quad\cr}$$
```

这里第二行的额外'`{}`'是为了告诉 TEX, '`+`'是一个二元运算。对左边长的方程特别推荐使用双行形式; 在这种情况下, 断点一般选在 = 好前面。

▸ **练习19.18**

排版下列列表公式:

$$\sum_{1\le j\le n}\frac{1}{(x_j-x_1)\ldots(x_j-x_{j-1})(x-x_j)(x_j-x_{j+1})\ldots(x_j-x_n)}$$
$$=\frac{1}{(x-x_1)\ldots(x-x_n)}. \qquad (27)$$

▸ **练习19.19**

如果必须在一个窄栏中输入象下面这样的巨大分数:

$$\frac{q^{\frac12 n(n+1)}(ea;q^2)_\infty(eq/a;q^2)_\infty(caq/e;q^2)_\infty(cq^2/ae;q^2)_\infty}{(e;q)_\infty(cq/e;q)_\infty}$$

可能你只能把分子裂分而变成

$$\frac{q^{\frac{1}{2}n(n+1)}(ea;q^2)_\infty(eq/a;q^2)_\infty \atop \hfill (caq/e;q^2)_\infty(cq^2/ae;q^2)_\infty}{(e;q)_\infty(cq/e;q)_\infty}$$

怎样用 TEX 得到后一个分数?

20. 定义（宏）

在 TEX 文稿中, 把经常用到的命令和符号的组合用控制系列来表示, 在输入时就可以节约时间了。例如, 如果在文档中用到很多‘$(x_1,\ldots,x_n)$’, 那么就可以输入

```
\def\xvec{(x_1,\ldots,x_n)}
```

这样就把 `\xvec` 定义为‘`(x_1,\ldots,x_n)`’了。因此, 象

$$\sum_{(x_1,\ldots,x_n)\ne(0,\ldots,0)} \bigl(f(x_1,\ldots,x_n)+g(x_1,\ldots,x_n)\bigr)$$

这样复杂的列表公式就可以简单地输入为

```
$$\sum_{\xvec\ne(0,\ldots,0)} \bigl(f\xvec+g\xvec\bigr)$$
```

而不需要用冗长的输入。通过定义象 `\xvec` 这样的控制系列, 不但可以减轻输入的工作量, 还可以在输入时减少出现错误和矛盾的机会。

当然, 通常不要单单为快速输入一个公式而给出定义; 这样没什么好处, 因为在你决定是否给出定义时, 以及输入定义本身时就花掉时间了。只有当整个文稿中用到某些符号组合的次数超过一打时, 才真正能得到好处。在排版前, 聪明的人会浏览一下整个文档, 对要出现的问题和要用到的定义有一个印象。例如, 第十六章建议, 对频繁使用符号 $\hat A$ 的文稿, 在其开头给出控制系列 `\Ahat` 的定义。

象 `\xvec` 这样的缩写在计算机的很多方面中都很有用, 并且因为其威力巨大而被称为*宏*; 一个很小的宏可以表示很多的内容, 因此它有一种宏观的效果。象 TEX 这样的系统程序, 它处理用户定义的方法被称为*展开*式; 例如, `\xvec` 展开为 `(x_1,\ldots,x_n)`, 而 `\ldots` 接着展开为 `\mathinner{\ldotp\ldotp \ldotp}`。因此, `\xvec` 实际上是‘`(x_1,\mathinner{\ldotp\ldotp\ldotp},x_n)`’的缩写。(展开到此为止, 因为 `\mathinner` 是 TEX 的原始控制系列, 而且 `\ldotp` 已经由 `\mathchardef` 定义好了; 因此, `\mathinner` 和 `\ldotp` 不是宏。)

TEX 用户一般要建立自己个人的宏库, 以便在不同的文档中使用。例如, 一般建立一个叫 `macros.tex` 的文件, 其中有你喜爱的专用控制系列的定义, 可能还有载入所喜欢的特殊字体的命令, 等等。如果在文稿开头输入命令:

```
\input macros
```

那么 TEX 将读入所有这些定义, 省得你再次输入它们了。当然, TEX 的内存有限, 读入文件要花时间, 因此不要在 `macros.tex` 中放上千个定义。一个大的宏定义集(比如, 附录 B 中的定义集)称为一个*格式*(比如, “plain TEX 格式”); 如果格式不经常改变, 那么 TEX 能用一种特殊的方式快速输入一个格式。

例子 `\xvec` 和 `\Ahat` 是应用于数学公式的, 但是即使与数学无关时, 宏定义也非常有用。例如, 如果要用 TEX 进行商业通信, 就可以用宏 `\yours` 了表示‘Sincerely yours, A. U. Thor’。如果经常用到格式信函,

就可以用宏生成整个句子或段落, 或者是一组段落。例如, 税务机关可以用下面两个宏:

```
\def\badcheck{A penalty has been added because your
  check to us was not honored by your bank.\par}
\def\cheater{A penalty of 50\% of the underpaid tax
  has been added for fraud.\par}
```

象这样简单的宏定义以'`\def`'开始; 接着是控制系列的名字, 比如'`\badcheck`'; 再接着是要替换的文本, 它们要封装在'`{`'和'`}`'中。在这种情况下, 大括号不表示编组; 它们直接表明定义中要替换的文本的范围。当然, 在替换文本中包含大括号的宏也是可以定义的, 只要这些大括号互相正确匹配即可。例如, '`\def\xbold{{\bf x}}`'把 `\xbold` 定义为'`{\bf x}`'。

▶ **练习20.1**

定义一个叫 `\punishment` 的宏, 它要把'I must not talk in class.'这句话输出 100 行。[提示:首先定义输出一次的宏 `\mustnt`, 再定义把 `\mustnt` 输出五次的宏。]

▶ **练习20.2**

下面给出的宏 `\puzzle` 的定义, 展开后是什么?

```
\def\a{\b}
\def\b{A\def\a{B\def\a{C\def\a{\b}}}}
\def\puzzle{\a\a\a\a\a}
```

只要你略微思考一下上面例子中的简单宏, 可能就会想: "小子, 如果宏展开后某些文本可以改变不是更好? 最好能把不同的东西放在那些文本中间。" 好, TEX 带来了好消息: 控制系列可以用参数的方式来定义, 并且你可以给出要代替参数的变量。

例如, 再来看看 `\xvec`。假定你不只需要'$(x_1,\ldots,x_n)$', 还需要频繁用到'$(y_1,\ldots,y_n)$'和其它类似的公式。那么你可能要输入

```
\def\row#1{(#1_1,\ldots,#1_n)}
```

此后, `\row x` 得到的就是'$(x_1,\ldots,x_n)$', `\row y` 得到的是'$(y_1,\ldots,y_n)$'。符号 `#1` 表示宏的第一个参数, 并且当你输入'`\row x`'时, `x` 就是所谓的变量, 它要代替的是替换文本中 `#1`。在这种情况下, 变量由单个字母 `x` 组成。你还可以使用 `\row\alpha`, 这时变量是一个控制系列, 所得到的结果是'$(\alpha_1,\ldots,\alpha_n)$'。如果要让变量包含不止一个符号或控制系列, 那么可以把它们封装在大括号中; 例如, `\row{x'}` 得到的是 $(x'_1,\ldots,x'_n)$。在这种情况下, 变量是 `x'` (没有大括号)。顺便说一句, 如果输入 `\row{{x'}}`, 那么得到的是 $({x'}_1,\ldots,{x'}_n)$; 原因是当取变量时, 只剥去一对大括号, 并且按照第十六章的规则, $({x'}_1,\ldots,{x'}_n)$ 是从 `({x'}_1,\ldots,{x'}_n)` 的数学模式得到的。

▶ **练习20.3**

继续讨论这个例子, `$\row{\bf x}$` 得到的是什么?

符号'#1'意味着可以有不止一个参数, 的确如此。例如, 可以输入

```
\def\row#1#2{(#1_1,\ldots,#1_#2)}
```

此后, '\row xn'就能得到'$(x_1,\ldots,x_n)$'了。至多可以有九个参数, #1 到 #9, 并且在使用时必须按顺序来。例如, 如果定义中没有叫做 #4 的前一个参数, 就不能用 #5。(这个限制只适用于替换文本开始前参数的初始声明; 在替换文本自己中, 所声明的参数可以任意次序地使用任意次。)

一个控制系列同时只能有一个定义, 因此, 如果在同一文稿中出现两个 \row 的定义, 那么第二个就替掉第一个。只要 TEX 遇见要展开的宏, 它就使用最近的定义。但是, 对于包含定义的组而言, 定义是局域的; 当组结束时, 旧定义将按照通常的方法恢复。

注意: 当象这些例子中用简单参数定义宏时, 必须注意不要在替换文本前面的'{'前留下空格。例如, '\def\row #1 #2 {...}'与'\def\row#1#2{...}'得到的结果是不同的, 因为 #1 和 #2 后的空格就要求 TEX 去寻找后面带空格的变量。(就象下面讨论的那样, 变量可以用相当一般的方法来"分界"。) 但是 \row 的空格象通常那样是任意的, 因为 TEX 总是忽略掉控制词后的空格。在给出'\def\row#1#2{...}'后, 就允许在变量之间放置空格(比如'\row x n'), 因为 TEX 不把单个空格当作未分界的变量。

下面的练习特别推荐给要掌握 TEX 宏的人。即使你略过其它的练习, 也要亲手做下面这个。

▶ **练习20.4**

扩展练习 20.1, 定义一个"适用更广的告诫"的宏, 它有两个参数, 使得 \punishment{run}{the halls} 得到的是 100 句'I must not run in the halls.'。

TEX 还允许定义这样的宏, 其参数用相当普遍的方法来分界; 你不需要总是把变量封装在大括号中。例如,

```
\def\cs #1. #2\par{...}
```

定义了一个控制系列 \cs, 它有两个参数, 并且这两个参数如下确定出: #1 由 \cs 和下一个随后出现的'.␣'(句点和空格)之间的所有内容组成; #2 由这个'.␣'和其后出现的 \par 之间的所有内容组成。(\par 可以明确给出, 也可以由第十八章讨论的空行生成。) 例如, 当 TEX 展开

```
\cs You owe \$5.00. Pay it.\par
```

时, 第一个变量是'You owe \$5.00', 而第二个是'Pay it.'。本例中, '\$5.00'中的句点不能结束 #1, 因为 TEX 要找的是后面紧跟着空格的句点。

还有, 如果变量的分界符被封装在大括号时, 这个变量不在此结束, 因为这样就得到了不匹配的大括号。例如, 在

```
\def\cs #1.#2\par{...}
```

中, 现在第一个变量到一个单句点处结束, 因此, 如果象上面那样调用 \cs, #1 为'You owe \$5', 而 #2 为'00. Pay it.'。但是,

```
\cs You owe {\$5.00}. Pay it.\par
```

圆满地把第一个句点藏起来了, 使得它不能结束变量 `#1`, 最后得到的是'`You owe {\$5.00}`'。

如果你要设计一个数学论文的格式, 那么可能要有陈述定理, 定义, 引理, 推论等东西的宏。例如, 可能要排版象下面这样的陈述:

Theorem 1. *TEX has a powerful macro capability.*

它用下列方法得到:

```
\proclaim Theorem 1. \TeX\ has a powerful macro capability.\par
```

实际上, plain TEX 中有一个象这样的宏 `\proclaim`; 其定义为

```
\def\proclaim #1. #2\par{\medbreak
  \noindent{\bf#1.\enspace}{\sl#2}\par\medbreak}
```

这样, 变量就象我们 `\cs` 的第一个例子那样分界。这里的替换文本用 `\medbreak` 把要 proclaim 的段落与其前后内容分开; proclaim 的题目设置为 bold 字体, 而文本设置为 slanted。(附录 B 中 `\proclaim` 的实际定义与它不完全一样; 为了使分页不出现在定理陈述紧后面, 而修改了最后的 `\medbreak`。因此, 短小的定理更倾向于放在页面顶部而不是底部。)

通过改变宏 `\proclaim`, 你可以改变论文中所有 proclaim 的格式, 而不需要改动论文自己的文本。例如, 可以得到象下面这样的东西:

THEOREM 1: *TEX has a powerful macro capability.*

这只需要直接修改 `\proclaim` 的替换文本即可, 我们假定你有所用到的字体。 TEX 希望支持复合的高级语言, 其中用户使用的所有的控制系列都是宏, 而不是 TEX 原始命令。思路是要能用几个要素来描述出重要类型的文档, 而不涉及到实际的字体, 字体尺寸或详细的间距; 因此, 可以用很多不同的字体来设置一个与字体无关的文档。

现在我们已经看到一些例子, 接下来我们讨论 TEX 宏的准确规则。定义的一般形式为

\def⟨control sequence⟩⟨parameter text⟩**{**⟨replacement text⟩**}**

其中 ⟨parameter text⟩ 不包含大括号, 并且在 ⟨replacement text⟩ 中所出现的 `{` 和 `}` 要正确嵌套。还有, 符号 `#` 有特殊含义: 在 ⟨parameter text⟩ 中, 第一个出现的 `#` 后面必须跟 `1`, 下一个要跟 `2`, 等等; 只允许九个 `#`。在 ⟨replacement text⟩ 中, 每个 `#` 后面必须跟一个在 ⟨parameter text⟩ 中的 `#` 后面出现过的数字, 或者一个 `#` 后面跟一个 `#`。表示当宏展开时, 后一种情况表示一个单个 `#`; 前一种情况表示插入相应的变量。

例如, 我们讨论一个"随便的"定义, 它只是为了展示 TEX 的规则。定义

```
\def\cs AB#1#2C$#3\$ {#3{ab#1}#1 c##\x #2}
```

表示, 控制系列 `\cs` 有一个参数文本, 它由 9 个记号组成(假定采用 plain TEX 的类代码):

$\mathtt{A}_{11}$, $\mathtt{B}_{11}$, `#1`, `#2`, $\mathtt{C}_{11}$, $\$_3$, `#3`, $\boxed{\$}$, $␣_{10}$

有一个替换文本, 由 12 个记号组成:

`#3`, $\{_1$, a_{11}, b_{11}, `#1`, $\}_2$, `#1`, $␣_{10}$, c_{11}, $\#_6$, [x], `#2`.

因此, 当 TEX 读入控制系列 `\cs` 后, 要读入的下两个记号应该是 A_{11} 和 B_{11} (否则就会得到错误信息'`Use of \cs doesn't match its definition`'); 接下来是变量 `#1`, 变量 `#2`, C_{11}, $\$_3$, 再接着是变量 `#3`, `\$`, 最后是空格记号。习惯上用"变量"这个词表示要替换参数的记号串; 参数出现在定义中, 而当使用定义时, 出现的是变量。(为此, 我们要扩展第七章记号的定义: 除了控制系列和(字符代码, 类代码)对外, TEX 还有"参数记号", 在这里它用 `#1` 到 `#9` 来 表示。参数记号只出现在宏的记号列中。

问: TEX 怎样才能知道变量在哪里结束? 答: 分两种情况。在 ⟨parameter text⟩ 中, *分界的参数*要跟一个或多个非参数记号, 才能到达参数文本的结尾或下一个参数记号; 在这种情况下, 相应的变量是长度最短(可能是空)的记号序列, 并且组的嵌套要正确, 后面跟的是在输入中的非参数记号列。(类代码和字符代码都要匹配, 而且控制系列命名必须相同。) 在 ⟨parameter text⟩ 中, *未分界的参数*后面要紧接一个参数记号, 或者要出现在参数文本的最结尾; 在这种情况下, 相应的变量就是下一个非空的记号, 如果此记号是'`{`', 这时变量是所跟的整个组 `{...}`。在所有两种情况下, 如果用这两种方法找到的的变量的形式为'`{`⟨nested tokens⟩`}`', 其中 ⟨nested tokens⟩ 表示关于大括号正确嵌套的任意记号列, 那么就去掉封装记号的最外层的盒子, 只留下 ⟨nested tokens⟩。例如, 我们继续讨论上面的例子 `\cs`, 假定接下来的文本包含

```
\cs AB {\Look}C${And\$ }{look}\$ 5.
```

变量 `#1` 将是记号 [Look], 因为 `#1` 是未分界的参数(在定义中, 它后面紧跟的是 `#2`); 在这种情况下, TEX 将忽略掉 `B` 后面的空格, 并且剥去 `{\Look}` 外面的大括号。变量 `#2` 将是空的, 因为 `C$` 紧跟着。而变量 `#3` 将是对应于文本 `{And\$␣}{look}` 的 13 个记号, 因为 `#3` 后面跟着'`\$␣`', 并且因为第一个出现的'`\$␣`'在大括号里面。即使变量 `#3` 以左大括号开头, 以右大括号结尾, 这些大括号也不会被剥去, 因为那样得到的是未嵌套的记号'`And\$ }{look`'。因此, 总的结果是, 在替换文本中用变量替换参数后, TEX 下一个要读入的记号列为

```
{And\$ }{look}{ab\Look}\Look␣c#\x5.
```

这里的空格 ␣ 是得到的记号列的一部分, 即使它跟在控制词 `\Look`, 因为只有当 TEX 首先把输入行转换为记号列时控制词后面的空格才去掉, 就象第八章讨论的那样。

▶ **练习20.5**

例子 `\cs` 的定义中在替换文本中包括一个 `##`, 但是在此例子中这样使用 `##` 没什么意义。给出一个 `##` 有意义的例子。

这些规则允许一个特殊的扩展: 如果 ⟨parameter text⟩ 的最后一个字符是 `#`, 使得这个 `#` 后面紧跟着 `{`, 那么看起来就好像 TEX 把 `{` 插入到参数文本的右结尾和替换文本的右结尾处。例如, 如果给出'`\def\a#1#{\hbox to #1}`', 那么在后面文本中的'`\a3pt{x}`'将展开为'`\hbox to 3pt{x}`', 因为 `\a` 的变量以左大括号为定界符。

在定义的 ⟨parameter text⟩ 中, 在第一个参数记号前面的记号要跟在控制系列后面; 实际上, 它们变成了控制系列名字的一部分。例如, 作者可能给出过

```
\def\TeX/{...}
```

而没有使用无斜线的 `\TeX` 的定义。这样, 每次要得到 TEX 标志时必须输入 `\TeX/`, 但是新定义的优点是, ‘`\TeX/`’的空格不被忽略掉。可以利用这种思路来定义在句子中使用的宏, 使得用户不必担心空格可能会被吃掉。

▶ **练习20.6**
定义一个控制系列 `\a` 使得 `\a{...}` 展开为 `\b{...}`, 并且使得如果 `\a` 后面没有紧跟一个左大括号 TEX 就给出错误信息。

当你第一次定义复杂的宏时, 结果常常与你所预想的不同, 即使 TEX 的规则不太复杂。如果不知道为什么 `\def` 不听你的话, 可以使用帮助: 你可以设置 `\tracingmacros=1`, 这样 TEX 就在它展开宏和读入宏的变量时在你的 log 文件中写入一些东西。例如, 如果在 TEX 处理上面的宏 `\cs` 时 `\tracingmacros` 是正值, 它就把下列四行放到 log 文件中:

```
\cs AB#1#2C$#3\$ ->#3{ab#1}#1 c##\x #2
#1<-\Look
#2<-
#3<-{And\$ }{look}
```

在上述所有规则中, ‘`{`’, ‘`}`’和‘`#`’表示 TEX 读入宏定义时记号列中类代码分别为 1, 2 和 6 的任意字符; plain TEX 表示编组和参数的这些特殊符号没有什么不能改变的。你甚至可以同时使用同类代码的几个不同字符。

▶ **练习20.7**
假定‘`[`’, ‘`]`’和‘`!`’的类代码分别为 1, 2 和 6, 就象 ‘`{`’, ‘`}`’和 ‘`#`’一样。看看下面定义的意思是什么:

```
\def\!!1#2![{!#]#!!2}
```

当展开‘`\! x{[y]][z}`’时得到的记号列是什么?

在实践中, 我们总是要犯错误的。而且最常见的排版错误就是在宏的变量中落掉一个‘`}`’或者多插入一个‘`{`’。如果在这种情况下 TEX 盲目地遵循规则, 就要因为要找到变量的结尾而不得不读入越来越多的记号。但是象实际使用中如此多变量, 只要变量中有一个这样的笔误就无法结束(唉); 因此, TEX 不得不一直读到文件结束, 或者(更可能的是)直到记号把计算机内存都塞满为止。不论哪种情况, 单个的排版失误将导致无法运行, 并且用户被迫重新开始。因此 TEX 还有另一个规则, 用它把这样的错误限制在它所在的段落中: *记号‘`\par`’不允许作为变量的一部分而出现*, 除非你明确告诉 TEX 可以使用 `\par`。只要 TEX 把 `\par` 作为变量的一部分而包括进来, 它就终止当前宏的展开, 并且给出信息说, 发现一个“失控的变量”。

如果你真要定义一个控制系列, 其变量允许出现记号 \par, 那么在'\def'紧前面加上'\long'把它定义为一个"长"宏。例如, 宏 \bold 的定义为

```
\long\def\bold#1{{\bf#1}}
```

它可以把几个段落都设置为 bold 字体。(但是, 这样的宏并不是排版 bold 文本的特别好的方法。更好的是, 比如

```
\def\beginbold{\begingroup\bf}
\def\endbold{\endgroup}
```

因为对长的变量, 它不会把 TeX 的内存塞满。)

但是, 禁用 \par 这个方法不能预防所有可能出现的落掉大括号的错误; 如果你在 \def 的结尾处落掉了 }, 也要出同样的问题。在这种情况下, 更难限制住错误, 因为 \par 在替换文本中是一个有用的东西; 在这里我们不想禁用 \par, 因此, TeX 提供了另一种方法: 当宏定义前面有'\outer'时, 相应的控制系列不允许出现在记号被高速读入的任何地方。\outer 宏不允许出现在变量中(即使允许出现 \par 它也不能出现), 也不允许出现在定义的参数文本或替换文本中, 以及对齐方式的前言中和要被跳过的条件文本中。如果 \outer 宏确实在这些地方出现了, 那么 TeX 就停下来, 并且给出"失控地点"或者一个"不完整"的条件。在此意义上, 输入文件的结束也被看作是 \outer 的; 例如, 文件不能在定义中间结束。如果你为他人设计了一个格式, 通过在每个在文档中只"默默运行"的控制系列前面加上 \outer, 就可以检验出可能出现的错误。例如, 附录 B 把 \proclaim 定义为 \outer 型, 因为用户不应该把定理陈述为定义, 变量或前言的一个部分。

我们现在已经知道, \def 前面可以加 \long 或 \outer, 并且如果定义要延伸到组外还可以在前面加 \global。这三个前缀可以按任意次序放在 \def 前面, 并且可以不止出现一次。 TeX 还有一个原始命令 \gdef, 它等价于 \global\def。因此, 例如,

```
\long\outer\global\long\def
```

与'\outer\long\gdef'的意思是一样的。

到现在为止, 在本手册我们已经见过几种定义控制系列的方法。例如,

\font\cs=⟨external font name⟩	把 \cs 变成字体标识符;
\chardef\cs=⟨number⟩	把 \cs 变成字符代码;
\countdef\cs=⟨number⟩	把 \cs 变成 \count 寄存器;
\def\cs...{...}	把 \cs 变成宏。

现在要给出此类中的另一个重要的命令:

\let\cs=⟨token⟩	把记号当前的含义赋予给 \cs。

如果 ⟨token⟩ 是另一个控制系列, \cs 就得到与那个控制系列同样的意义。例如, 如果给出'\let\a=\def', 那么用'\a\b...{...}'就可以定义宏 \b, 因为 \a 的性质象 TeX 的原始命令 \def 一样。如果我们给出

```
\let\a=\b \let\b=\c \let\c=\a
```

就把 `\b` 和 `\c` 中的原来的意思互换了。并且如果给出

```
\outer\def\a#1.{#1:}
\let\b=\a
```

其结果与'`\outer\def\b#1.{#1:} \let\a=\b`'完全一样。

如果 `\let` 中的 ⟨token⟩ 是单个字符——即如果它是(字符代码, 类代码)对——那么在一定程度上控制系列就象此字符一样; 但是有一些不同。例如, 给出'`\let\zero=0`'后, 不能在数字常数中使用 `\zero`, 因为 TEX 要求宏展开后数字常数中的记号是数字; `\zero` 不是一个宏, 因此它不能展开。但是, 这样的 `\let` 也有其用处, 我们将在后面讨论。

▶ **练习20.8**
'`\let\a=\b`'和'`\def\a{\b}`'之间有无明显的不同?

▶ **练习20.9**
实际运行 TEX 试着得到下面问题的答案: (a). 如果控制系列 `\par` 被重新定义(比如, '`\def\par{\endgroup\par}`'), 那么 `\par` 在变量中仍然被禁用吗? (b). 如果给出了 `\let\xpar=\par`, 那么 `\xpar` 在变量中也被禁用吗?

TEX 也允许'`\futurelet\cs`⟨token$_1$⟩⟨token$_2$⟩'这种结构, 与'`\let\cs =` ⟨token$_2$⟩⟨token$_1$⟩⟨token$_2$⟩'的结果一样。其想法是, 比如, 你可以在宏的替换文本结尾处使用'`\futurelet\a\b`'; TEX 将把 `\a` 设置为此宏后面的记号, 其后再展开 `\b`。控制系列 `\b` 将继续处理, 并且它可以检验 `\a` 来发现接下来出现的是什么。 ???

在学习了含参数的宏之后, 要做的下一件事情就是编写宏, 在当前条件下, 用它来改变排版的方式。为此, TEX 提供了各种原始命令。其中"条件文本"的一般形式为

`\if`⟨condition⟩⟨true text⟩`\else`⟨false text⟩`\fi`

其中, 如果 ⟨condition⟩ 为假, 那么跳过 ⟨true text⟩; 如果 ⟨condition⟩ 为真, 那么跳过 ⟨false text⟩。如果 ⟨false text⟩ 为空, 就可以省略掉 `\else`。这个结构'`\if`⟨condition⟩'的部分以一个控制系列开头, 其前两个字母是'`if`'; 例如,

```
\ifodd\count0 \rightpage \else\leftpage \fi
```

规定了一个条件, 当 TEX 的整数寄存器 `\count0` 是奇数时为真。因为 TEX 一般把当前页码放在 `\count0`, 所以在本例中, 如果页码是奇数, 将展开宏 `\rightpage`, 而如果页码是偶数就展开 `\leftpage`。条件命令总以最后的'`\fi`'来结束。

条件文本主要是为有经验的 TEX 用户编写高级宏而准备的; 因此, 本章剩下的段落前面都有双"危险"标志。从这里跳到第二十一章不用内疚; 换句话说, 假想本手册在这里使用了'`\ifexperienced`', 而所匹配的'`\fi`'就出现在本章的结尾。

在我们讨论 TEX 的 \if... 命令的指令集前，来看另一个例子，这样一般思路就清楚了。假定 \count 寄存器 \balance 存放的是某人付所得税后的余额；这个量用美分来表示，并且它可以是正的，负的或者是零。我们的简单目的就是编写一个宏，它要为税务机关按照 balance 的值生成一份报告，此报告要作为通知书的一部分寄给此人。正 balance 的报告与负的差别很大，因此我们可以用 TEX 的条件文本来做：

```
\def\statement{\ifnum\balance=0 \fullypaid
  \else\ifnum\balance>0 \overpaid
       \else\underpaid
       \fi
  \fi}
```

这里，\ifnum 是一个比较两个数的条件命令；如果 balance 为零，宏 \statement 就只剩下 \fullypaid，等等。

对这个结构中 0 后面的空格要特别注意。如果例子中的给出的是

```
...=0\fullypaid...
```

那么 TEX 在得到常数 0 的值之前要展开‘\fullypaid’，因为 \fullypaid 可能以 1 或其它东西开头而改变这个数字。(毕竟，在 TEX 看来，‘01’是完全可以接受的 ⟨number⟩。) 在这种特殊情况下，程序照样工作，因为待会我们就可以看到，\fullypaid 的开头是字母 Y；因此，落掉空格后唯一引起的问题就是 TEX 处理变慢，因为它要跳过的是整个展开的 \fullypaid，而不仅仅是一个未展开的单个记号 \fullypaid。但是在其它情形，象这样落掉空格可能使得 TEX 在你不希望展开宏时把它展开，并且这样的反常会得到敏感而混乱的错误。要得到最好的结果，*总是要在数值常数后面放一个空格*；整个空格就告诉 TEX 这个常数是完整的，而这样的空格从来不会程序在输出中。实际上，当不在常数后面放空格时，TEX 实际上要做更多的事，因为每个常数都要持续到读入一个非数字字符为止；如果这个非数字字符不是空格，TEX 就把你的确有的记号取出来并且备份，以便下次再读。(另一方面，当某些其它字符紧跟常数时，作者常常忽略掉空格，因为额外的空格在文件中挺难看的；美感比效率更重要。)

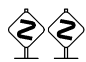
▶ **练习20.10**

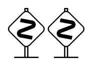
继续看看税务机关的例子，假定 \fullypaid 和 \underpaid 的定义如下：

```
\def\fullypaid{Your taxes are fully paid---thank you.}
\def\underpaid{{\count0=-\balance
  \ifnum\count0<100
    You owe \dollaramount, but you need not pay it, because
    our policy is to disregard amounts less than \$1.00.
  \else Please remit \dollaramount\ within ten days,
    or additional interest charges will be due.\fi}}
```

按此编写宏 \overpaid，假定 \dollaramount 是一个宏，它按美元和美分输出的 \count0 的内容。你的宏应该给出的是：a check will be mailed under separate cover, unless the amount is less than $1.00, in

which case the person must specifically request a check.


▶ **练习20.11**
编写宏 `\dollaramount`, 这样税务机关的 `\statement` 就完整了。


现在, 我们完整地总结一下 TEX 的条件命令。其中有一些本手册还未讨论的东西。

- `\ifnum`⟨number$_1$⟩⟨relation⟩⟨number$_2$⟩ (比较两个整数)

⟨relation⟩ 编写必须是‘`<`$_{12}$’, ‘`=`$_{12}$’或‘`>`$_{12}$’。两个整数按通常的方法进行比较, 因此所得结果为真或假。

- `\ifdim`⟨dimen$_1$⟩⟨relation⟩⟨dimen$_2$⟩ (比较两个尺寸)

它与 `\ifnum` 类似, 但是比较的是两个 ⟨dimen⟩ 的值。例如, 要检验 `\hsize` 的值是否超过 100 pt, 可以用‘`\ifdim\hsize>100pt`’。

- `\ifodd`⟨number⟩ (奇数测试)

如果 ⟨number⟩ 为奇数则真, 为偶数则假

- `\ifvmode` (垂直模式测试)

如果 TEX 处在垂直模式或者内部垂直模式则真(见第十三章)。

- `\ifhmode` (水平模式测试)

如果 TEX 处在水平模式或受限水平模式则真(见第十三章)。

- `\ifmmode` (数学模式测试)

如果 TEX 处在数学模式或列表数学模式则真(见第十三章)。

- `\ifinner` (内部模式测试)

如果 TEX 处在内部垂直模式或受限水平模式或(非列表)数学模式则真(见第十三章)。

- `\if`⟨token$_1$⟩⟨token$_2$⟩ (测试字符代码是否相同)

TEX 将把 `\if` 后面的宏展开为两个不能再展开的记号。如果其中一个记号是控制系列, 那么 TEX 就把它看作字符代码为 256 和类代码为 16, 除非此控制系列的当前内容被 `\let` 为等于非活动字符记号。用这种方法, 每个记号给出一个(字符代码, 类代码)对。如果字符代码相等, 条件成立, 而与类代码无关。例如, 在给出 `\def\a{*}`, `\let\b=*` 和 `\def\c{/}` 后, ‘`\if*\a`’和‘`\if\a\b`’为真, 但是‘`\if\a\c`’为假。还有, ‘`\if\a\par`’为假, 但是‘`\if\par\let`’为真。

- `\ifcat`⟨token$_1$⟩⟨token$_2$⟩ (测试类代码是否相同)

它就象 `\if` 那样, 但是测试的是类代码, 而不是字符代码。活动字符的类代码是 13, 但是为了让 `\if` 或 `\ifcat` 在得到这样的字符时强制展开, 必须给出‘`\noexpand`⟨active character⟩’。例如, 在

```
\catcode`[=13 \catcode`]=13 \def[{*}
```

之后, 测试‘`\ifcat\noexpand[\noexpand]`’和‘`\ifcat[*`’为真, 而测试‘`\ifcat\noexpand[*`’为假。

- \ifx⟨token$_1$⟩⟨token$_2$⟩ (测试记号是否相同)

在这种情况下, 当 TEX 遇见这两个记号时, 不展开控制系列。如果(a) 两个记号不是宏, 并且它们都标识同一(字符代码, 类代码)对或同一 TEX 原始命令, 同一 \font 或 \chardef 或 \countdef 等等; (b) 两个记号是宏, 并且它们对 \long 和 \outer 都处在相同的状态, 以及它们有同样的测试和"顶级"展开, 那么条件为真。例如, 设置'\def\a{\c} \def\b{\d} \def\c{\e} \def\d{\e} \def\e{A}'后, 在 \ifx 测试中, \c 和 \d 相等, 但是 \a 和 \b, \d 和 \e 不相等, \a, \b, \c, \d, \e 的其它组合也不相等。

- \ifvoid⟨number⟩, \ifhbox⟨number⟩, \ifvbox⟨number⟩ (测试一个盒子寄存器)

⟨number⟩ 应该在 0 和 255 之间。如果 \box 是置空的, 或者包含一个 hbox 或 vbox 时条件为真(见第十五章)。

- \ifeof⟨number⟩ (测试文件是否结束)

⟨number⟩ 一个在 0 和 15 之间。条件为真, 除非相应的输入流是开的并且没有读完。(见下面的命令 \openin。)

- \iftrue, \iffalse (永远为真或为假)

这些条件有预先确定的结果。但是它们却非常有用, 见下面的讨论。

最后, 有一个多条件结构, 它与其它的不同, 因为它有多个分支:

- \ifcase⟨number⟩⟨text for case 0⟩\or⟨text for case 1⟩\or ⋯
  \or⟨text for case n⟩\else⟨text for all other cases⟩\fi

在这里, $n+1$ 种情形由 n 个 \or 分开, 其中 n 可以是任意非负数。⟨number⟩ 选择要使用的文本。\else 部分还是可选的, 如果你不想在 ⟨number⟩ 为负数或大于 n 的情形下给出某些文本的话。

▶ **练习20.12**
设计一个宏 \category, 在其后面输入单个字符后, 它用符号输出此字符的当前类代码。例如, 如果使用 plain TEX 的类代码, '\category\\'将展开为'escape', '\category\a'将展开为'letter'。

▶ **练习20.13**
用下列问题测验一下自己, 看看你是否掌握了这些模糊的情形: 在设置定义'\def\a{} \def\b{**} \def\c{True}'后, 下面哪些是真的? (a) '\if\a\b'; (b) '\ifcat\a\b'; (c) '\ifx\a\b'; (d) '\if\c'; (e) '\ifcat\c'; (f) '\ifx\ifx\ifx'. (g) '\if\ifx\a\b\c\else\if\a\b\c\fi\fi'.

注意, 所有的条件控制系列都以 \if... 开头, 并且它们都要匹配 \fi。这种 \if... 与 \fi 配对的约定使程序中条件控制系列的嵌套更好分清。\if...\fi 的嵌套与 {...} 的嵌套无关; 因此, 可以在条件控制系列中间开始或结束一个组, 也可以在组中间开始或结束一个条件控制系列。编写宏的大量经验表明, 这种无关性在应用中很重要; 但是如果不仔细也会出现问题。

有时候要把信息从一个宏传到另一个, 而实现它有几种方法: 把它作为一个变量来传递, 把它放在一个寄存器中, 或者定义一个包含此信息的控制系列。例如, 附录 B 中的宏 \hphantom,

\vphantom 和 \phantom 非常相似, 因此作者希望把它们三个的所有部分放在另一个宏 \phant 中。用某种方法来告诉 \phant 所要的是哪类 phantom。第一种方法是定义象下面这样的控制系列 \hph 和 \vph:

```
\def\hphantom{\ph YN} \def\vphantom{\ph NY} \def\phantom{\ph YY}
\def\ph#1#2{\def\hph{#1}\def\vph{#2}\phant}
```

之后 \phant 可测试'\if Y\hph'和'\if Y\vph'。这可以用, 但是有几个更有效的方法; 例如, '\def\hph{#1}'可以用'\let\hph=#1'来代替, 以避免展开宏。因此, 一个更好的方法是:

```
\def\yes{\if00} \def\no{\if01}
\def\hphantom{\ph\yes\no}...\def\phantom{\ph\yes\yes}
\def\ph#1#2{\let\ifhph=#1\let\ifvph=#2\phant}
```

之后 \phant 可测试'\ifhph'和'\ifvph'。(这种结构出现在 TEX 语言中有 \iftrue 和 \iffalse 之前。) 想法很好, 因此作者就开始把 \yes 和 \no 用在其它情形中。但是接着有一天, 一个复杂的条件控制系列失败了, 因为它把象 \ifhph 这样的测试放在另一个条件文本中了:

```
\if...  \ifhph...\fi  ... \else  ...  \fi
```

能看出问题吗? 当执行最外层条件的 ⟨true text⟩ 时, 所有的都很顺利, 因为 \ifhph 是 \yes 或 \no, 并且它展开为 \if00 或 \if01。但是当跳过 ⟨true text⟩ 时, \ifhph 没有被展开, 因此第一个 \fi 错误地匹配到第一个 \if 上; 很快错误就都出来了。这时 \iftrue 和 \iffalse 就被加进 TEX 语言中, 来代替 \yes 和 \no; 现在, \ifhph 是 \iftrue 或 \iffalse, 因此不管它是否被跳过, TEX 都可正确匹配上 \fi。

为了便于构造 \if..., plain TEX 提供了一个叫 \newif 的宏, 这样在给出'\newif\ifabc'后, 就定义了三个控制系列: \ifabc(测试真假), \abctrue(测试为真)和 \abcfalse (测试为假)。现在, 附录 B 的 \phantom 问题就可以如下解决:

```
\newif\ifhph \newif\ifvph
\def\hphantom{\hphtrue\vphfalse\phant}
```

并且有 \vphantom 和 \phantom 的类似定义。不再需要宏 \ph 了; 还是 \phant 来测试 \ifhph 和 \ifvph。附录 E 中有由 \newif 生成的其它条件文本的例子。新的条件文本开始都设定为假。

注意: 不要在条件文本中给出象'\let\ifabc=\iftrue'这样的东西。如果 TEX 跳过这些命令, 就会认为 \ifabc 和 \iftrue 都要匹配一个 \fi, 因为 \let 没有被执行! 把这样的命令包在宏中, 这样 TEX 只有在不跳过要读入的文本时才能遇见'\if...'。

TEX 有 256 个"记号列寄存器"叫做 \toks0 到 \toks255, 因此记号列可以在不经过 TEX 读入器时很容易地传来传去。还有一个 \toksdef 指令, 使得, 比如,

```
\toksdef\catch=22
```

把 \catch 与 \toks22 等价起来。Plain TEX 提供了一个宏 \newtoks, 由它来分配新的记号列寄存器; 它类似于 \newcount。记号列寄存器的性质就象记号列参数 \everypar, \everyhbox, \output, \errhelp 等

等。为了给记号列参数或寄存器指定新值, 可以使用

⟨token variable⟩={⟨replacement text⟩}
或者 ⟨token variable⟩=⟨token variable⟩

其中 ⟨token variable⟩ 表示一个记号列参数, 或者是由 `\toksdef` 或 `\newtoks` 定义的一个控制系列, 或者是一个明确的寄存器名字'`\toks`⟨number⟩'。

经常使用宏这个便利工具的每个人都会遇到编写的宏出问题的时候。我们已经说过, 为了看看 TeX 是怎样展开宏和它找到的变量是什么, 我们可以设置 `\tracingmacros=1`。还有另一个有用的方法来看看 TeX 在做什么: 如果设置 `\tracingcommands=1`, 那么 TeX 将显示出它所执行的每个命令, 就象第十三章那样。还有, 如果设置 `\tracingcommands=2`, 那么 TeX 将显示所有条件命令及其结果, 以及实际执行或展开的非条件命令。这些诊断信息出现在 log 文件中。如果还设置了 `\tracingonline=1`, 那么在终端上也可以看到。(顺便说一下, 如果设置 `\tracingcommands` 大于 2, 那么得到的信息同等于 2 一样。) 类似地, `\tracingmacros=2` 将跟踪 `\output`, `\everypar`, 等等。

要知道宏命令出毛病的一个方法就是用刚才讨论的跟踪方法, 这样就能看到 TeX 每步在做什么。另一种就是掌握宏是怎样展开的; 现在我们来讨论这个规则。

TeX 的咀嚼过程把你的输入变成一个长记号列, 就象第八章讨论的那样; 其消化过程严格按照这个记号列进行。当 TeX 遇见记号列中的控制系列时, 要查找其当前的意思, 并且在某些情况下, 在继续读入之前要把此记号展开为一系列其它记号。展开过程作用的对象是宏和某些其它特殊的原始命令, 象 `\number` 和我们刚刚讨论过的 `\if` 这样。但是有时候, 却不进行展开; 例如, 当 TeX 处理一个 `\def`, 此 `\def` 的 ⟨control sequence⟩, ⟨parameter text⟩ 和 ⟨replacement text⟩ 不被展开。类似地, `\ifx` 后面的两个记号也不被展开。本章后面要给出一个完整列表, 在这些情况下不展开记号; 在实在没办法时, 你可以用它作为参考。

现在我们来看看不禁止展开时控制系列的展开情况。这样的控制系列分几种:

- 宏: 当宏被展开时, TeX 首先确定其变量(如果有的话), 就象本章前面讨论的那样。每个变量是一个记号列; 但记号作为变量而看待时, 它们不被展开。接着 TeX 用替换文本代替宏及其变量。

- 条件文本: 当 `\if...` 被展开时, TeX 读入必要的内容来确定条件的真假; 如果是假, 将跳过前面(保持 `\if...\fi` 的嵌套)直到找到要结束所跳过文本的 `\else`, `\or` 或 `\fi`。类似地, 当 `\else`, `\or` 或 `\fi` 被展开时, TeX 读入要跳过的任何文本的结尾。条件文本的"展开"是空的。(条件文本总是减少在后面的消化阶段所遇见的记号量, 而宏一般增加记号的量。)

- `\number`⟨number⟩: 当 TeX 展开 `\number` 时, 它读入所跟的 ⟨number⟩ (如果需要就展开记号); 最后的展开由此数的小数表示组成, 如果是负的前面要有'`-`'。

- `\romannumeral`⟨number⟩: 它与 `\number` 类似, 但是展开由小写 roman 数字组成。例如, '`\romannumeral 1984`'得到的是'`mcmlxxxiv`'。如果数字是零或负, 展开为空。

■ \string⟨token⟩: TEX 首先读入 ⟨token⟩ 而不展开。如果控制系列记号出现, 那么它的 \string 展开由控制系列的名字组成(如果控制系列不单单是活动字符, 就把 \escapechar 包括进来作为转义符)。否则, ⟨token⟩ 就是字符记号, 并且其字符代码保持为展开后的结果。

■ \jobname: 展开为 TEX 为此进程选定的名字。例如, 如果 TEX 把它的输出放在文件 paper.dvi 和 paper.log 中, 那么 \jobname 就展开为'paper'。

■ \fontname⟨font⟩: 展开为对应于所给定字体的外部文件名; 比如, '\fontname\tenrm'将展开为'cmr10'(五个记号)。如果字体所用的不是其设计尺寸, 那么"at size"也出现在展开中。⟨font⟩ 是一个由 \font 定义的标识符; 或是 \textfont⟨number⟩, \scriptfont⟨number⟩ 或 \scriptscriptfont⟨number⟩; 或者是 \font, 它表示当前字体。

■ \meaning⟨token⟩: TEX 把它展开为一系列字符, 它们是命令'\let\test=⟨token⟩ \show\test'在你的终端上显示的内容。例如, '\meaning A'一般展开的是'the letter A'; 在'\def\A#1B{\C}'后, '\meaning\A'展开的是'macro:#1B->\C '。

■ \csname...\endcsname: 当 TEX 展开 \csname, 它要读入匹配的 \endcsname, 如果需要就展开记号; 在此展开中, 只有字符记号或保留下来。因此, 整个 \csname...\endcsname 文本的"展开"将是单个控制系列记号, 如果它当前没有定义, 则定义为 \relax。

■ \expandafter⟨token⟩: TEX 首先读入 \expandafter 紧后面的记号, 而不展开它; 我们称此记号为 t。接着, TEX 读入 t 后面的记号(如果此记号有一个变量, 那么可能有更多的记号), 把它用其展开代替。最后 TEX 把 t 放在此展开的前面。

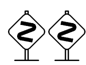
■ \noexpand⟨token⟩: 展开为记号自己; 但是如果此记号是一个按 TEX 的展开规则一般要被展开的控制系列, 那么其含义与'\relax'一样。

■ \topmark, \firstmark, \botmark, \splitfirstmark, 和 \splitbotmark: 展开为相应"标记"寄存器中的记号列(见第二十三章)。

■ \input⟨file name⟩: 展开为空; 但是 TEX 做好准备从给定文件的读入内容, 之后再在当前文件中继续读入其它记号。

■ \endinput: 展开为空。TEX 到达了 \input 行的结尾, 将停止从包含此行的文件中读入。

■ \the⟨internal quantity⟩: 展开为一列记号, 它表示某个 TEX 变量的当前值, 讨论见下面。例如, '\the\skip5'将展开为'5.0pt plus 2.0fil'(17 个记号)。



\the 这个有用的命令有很多子情形, 因此我们要同时讨论它们。各种内部数字量都可以被提出来使用:

■ \the⟨parameter⟩, 其中 ⟨parameter⟩ 是某个 TEX 整数参数(比如, \the\widowpenalty), 尺寸参数(比如, \the\parindent), 粘连参数(比如, \the\leftskip), 或 muglue 参数(比如, \the\thinmuskip)的名字。

■ \the⟨register⟩, 其中 ⟨register⟩ 是某个 TEX 的整数寄存器(比如, \the\count0), 尺寸寄存器(比如, \the\dimen169), 粘连寄存器(比如, \the\skip255), 或 muglue 寄存器(比如, \the\muskip\count2)的名字。

- \the⟨codename⟩⟨8-bit number⟩, 其中, ⟨codename⟩ 表示 \catcode, \mathcode, \lccode, \uccode, \sfcode 或 \delcode。例如, \the\mathcode`/ 得到的是斜线的当前数学代码(整数)。

- \the⟨special register⟩, 其中, 特殊寄存器为某个整数量 \prevgraf, \deadcycles, \insertpenalties, \inputlineno, \badness 或 \parshape(它只表示 \parshape 的行的数目); 或者是某个尺寸 \pagetotal, \pagegoal, \pagestretch, \pagefilstretch, \pagefillstretch, \pagefilllstretch, \pageshrink, \pagedepth。在水平模式下还可以指向一个特殊整数 \the\spacefactor, 在垂直模式下可用于一个特殊尺寸 \the\prevdepth。

- \the\fontdimen⟨parameter number⟩⟨font⟩, 它得到的是一个尺寸; 例如, 字体的参数 6 为其“em”的值, 因此, ‘\the\fontdimen6\tenrm’得到的是‘10.0pt’(6 个记号)。

- \the\hyphenchar⟨font⟩, \the\skewchar⟨font⟩, 它们得到的是定义给定字体的相应整数值。

- \the\lastpenalty, \the\lastkern, \the\lastskip, 它们得到的是当前列中最后一个项目的 penalty, kerning, 粘连或 muglue 的量, 如果此项目分别是 penalty, kern, 或粘连的话; 否则得到的是‘0’或‘0.0pt’。

- \the⟨defined character⟩, 其中 ⟨defined character⟩ 是一个控制系列, 它已经由 \chardef 或 \mathchardef 给定一个整数值; 结果就是此整数值, 用小数表示。

在到现在为止的所有情况下, \the 得到的结果是一系列 ASCII 字符记号。除了字符代码为 32 的记号(“空格”)的类代码为 10 外, 每个记号的类代码都是 12 (“其它字符”)。同样的类代码规则也适用于由 \number, \romannumeral, \string, \meaning, \jobname 和 \fontname 得到的记号。

在一些情形下, \the 得到的是非字符的记号, 是象 \tenrm 这样的字体标识符, 或者是任意记号列:

- \the⟨font⟩ 得到的是选择给定字体的字体标识符。例如, ‘\the\font’是对应于当前字体的控制系列。

- \the⟨token variable⟩ 得到的是一个变量当前值的记号列。例如, 可以展开‘\the\everypar’和‘\the\toks5’。

TEX 的原始命令‘\showthe’将把在展开定义中‘\the’所得到的东西显示在终端上; 展开前面加上‘> ’, 后面跟上句点。例如, 如果采用 plain TEX 的段落缩进, 那么‘\showthe\parindent’显示的是

```
> 20.0pt.
```

下面是以前说过的可展开的记号不被展开的所有情形的列表。某些情形中含有未讨论过的原始命令, 但是我们最后会给出它们的。在下列情形下展开被禁止:

- 当记号在错误修复期间被删除时(见第六章)。
- 当因为条件文本被忽略而记号被跳过时。
- 当 TEX 正在读入宏的变量时。
- 当 TEX 正在读入由下列定义的控制系列时: \let, \futurelet, \def, \gdef, \edef, \xdef, \chardef, \mathchardef, \countdef, \dimendef, \skipdef, \muskipdef, \toksdef, \read 和 \font。

- 当 TeX 正在读入下列控制系列的变量记号时: `\expandafter`, `\noexpand`, `\string`, `\meaning`, `\let`, `\futurelet`, `\ifx`, `\show`, `\afterassignment`, `\aftergroup`。
- 当 TeX 正在读入的是 `\def`, `\gdef`, `\edef` 或 `\xdef` 的参数文本时。
- 当 TeX 正在读入的是 `\def` 或 `\gdef` 或 `\read` 的替换文本; 或者是象 `\everypar` 或 `\toks0` 这样的记号变量的文本; 或者是 `\uppercase` 或 `\lowercase` 或 `\write` 的记号列。(当 `\write` 的记号列实际输出到一个文件时要被展开。)
- 当 TeX 正在读入对齐的前言时, 但是除了在原始命令 `\span` 的一个记号后, 或者正在读入 `\tabskip` 的 ⟨glue⟩ 时。
- 在数学模式开始的记号 `$`$_3$ 紧后面时, 这是为了看看是否后面还跟着一个 `$`$_3$。
- 在开始字母常数的记号 `` ` ``$_{12}$ 的后面。

有时候你会发现自己要定义一个新宏, 它的替换文本由于当前情形而已经被展开了, 而不是简单地逐字复制替换文本。为此, TeX 提供了命令 `\edef` (被展开的定义), 以及 `\xdef` (它等价于 `\global\edef`)。其一般格式与 `\def` 和 `\gdef` 一样, 但是 TeX 盲目地按照上面的展开规则来展开替换文本的记号。例如, 看看下面这个定义:

```
\def\double#1{#1#1}
\edef\a{\double{xy}} \edef\a{\double\a}
```

这里, 第一个 `\edef` 等价于'`\def\a{xyxy}`', 而第二个等价于'`\def\a{xyxyxyxy}`'。所有其它类型的展开也要进行, 包括条件文本; 例如, 在 TeX 给出 `\edef` 时处在数学模式的情况下

```
\edef\b#1#2{\ifmmode#1\else#2\fi}
```

的结果等价于'`\def\b#1#2{#1}`', 否则结果等价于'`\def\b#1#2{#2}`'。

由 `\edef` 和 `\xdef` 给出的被展开的定义要把记号展开到只剩下不能展开的记号, 但是由'`\the`'生成的记号列不再进一步展开。还有, '`\noexpand`'后面的记号不展开, 因为它的展开能力无效了。这两个命令可以用来控制展开哪些, 不展开哪些。

例如, 假设你要把 `\a` 定义为 `\b` (展开的)后面跟 `\c` (不展开) 再后面跟 `\d` (展开的), 并且假定 `\b` 和 `\d` 是无参数的简单宏。就可以用两种方法来实现:

```
\edef\a{\b\noexpand\c\d}
\toks0={\c} \edef\a{\b\the\toks0 \d}
```

甚至可以不用 `\noexpand` 或 `\the` 也可得到同样的效果; 对想多学习一些 TeX 展开原理的读者, 建议做一下下面三个练习。

▶ **练习20.14**

不用'`\noexpand`'和'`\the`', 找出定义前一段中 `\a` 的方法。

▶ **练习20.15**
继续讨论避免展开的例子, 假设要把 `\b` 完全展开到只剩下不能展开的记号, 但是你根本不希望展开 `\c`, 并且要把 `\d` 只展开一层。例如, 在定义 `\def\b{\c\c}`, `\def\c{*}` 和 `\def\d{\b\c}` 后, 要得到的是 `\def\a{**\c\b\c}` 这样的结果。怎样才能用 `\the` 来实现这个部分展开?

▶ **练习20.16**
不用 `\the` 或 `\noexpand` 来解决上一个练习。(这个练习很难。)

TEX 的原始命令 `\mark{...}`, `\message{...}`, `\errmessage{...}`, `\special{...}` 和 `\write`⟨number⟩`{...}` 都展开大括号中的记号列, 同 `\edef` 和 `\xdef` 基本一样。但是象 `#` 这样的宏参数字符在这样的命令中不用重复; 在 `\edef` 中用 `##`, 而在 `\mark` 中只用 `#`。命令 `\write` 有点特殊, 因为它的记号列首先读入而不展开; 直到记号实际上被写入一个文件时才进行展开。

▶ **练习20.17**
比较下面两个定义:

```
\def\a{\iftrue{\else}\fi}
\edef\b{\iftrue{\else}\fi}
```

哪个得到未匹配的左括号? (有点技巧。)

TEX 可以同时从大约 16 个文件中读入各个文本行, 除了在 `\input` 后的文件之外。为了开始读入这样一个辅助文件, 应该使用

`\openin`⟨number⟩=⟨file name⟩

其中 ⟨number⟩ 在 0 和 15 之间。(Plain TEX 用命令 `\newread` 了分配输入流的数字 0 到 15, 它类似于 `\newbox`。) 在大多数 TEX 的安装时, 如果没有明确给出扩展名, 那么扩展名'`.tex`'将添加到文件名之后, 就象用 `\input` 一样。如果文件找不到, TEX 不给出错误信息; 它只把输入流看作没有打开, 并且你可以用 `\ifeof` 来测试这种状态。当不再使用某个文件时, 可以使用

`\closein`⟨number⟩

并且如果与输入流数字相对应的文件是打开的, 那么它将关闭, 即, 返回其初始状态。为了从一个打开的文件得到输入, 可以使用

`\read`⟨number⟩`to`⟨control sequence⟩

并且所定义的控制系列是一个无参数的宏, 其替换文本是从指定文件读入的下一行的内容。此行用第八章的程序按当前类代码转换为一个记号列。如果需要, 再读入其它行, 直到左右大括号的数目相同。 TEX 在要读入的文件结尾暗中添加了一个空行。如果 ⟨number⟩ 不在 0 和 15 之间, 或者如果这样的文件没有打开, 或者文件结束了, 那么输入就来自终端; 如果 ⟨number⟩ 不是负值, 那么 TEX 将给出提示符。如果你不使用 `\global\read`, 那么宏的定义将是局域的。

例如, 通过命令 \read 以及 \message (它把一个展开的记号列写在终端上和 log 文件中), 可以很容易实现与用户对话:

```
\message{Please type your name:}
\read16 to\myname
\message{Hello, \myname!}
```

在这种情况下, 命令 \read 将写入'\myname='并等待应答; 应答在 log 文件中重复出来。如果'\read16'变成'\read-1', 那么'\myname='就被省略了。

▶ **练习20.18**
刚刚给出的 \myname 例子效果并不好, 因为在行尾的 ⟨return⟩ 被转换为一个空格。看看怎样解决这个小问题?

▶ **练习20.19**
继续看前一个例子, 定义一个叫 \MYNAME 的宏, 它包含所有 \myname 的字母, 但是用的是大写字母。例如, 如果 \myname 展开为 Arthur, 那么 \MYNAME 展开就是 ARTHUR。假定在 \myname 的展开中只包含字母和空格。

附录 B, D, E 包含大量编写的宏的例子, 可以做很多事情。现在, 在本章结尾, 我们通过几个例子来说明作为编程语言, TEX 实际上可以怎样使用, 如果你要得到某些特殊的效果, 并且不在意计算机所耗的时间。

Plain TEX 中含有一个 \loop...\repeat 结构, 它象这样工作: 给出'\loop α \if... β \repeat', 其中 α 和 β 是任意系列的命令, 并且 \if... 是任意条件测试(无匹配的 \fi)。 TEX 就首先执行 α; 接着如果条件为真, TEX 就执行 β, 并且再次从 α 开始重复整个过程。如果条件为假, 循环就停止。例如, 下面有一个小程序, 进行一段对话, 其中 TEX 等待用户输入'Yes'或'No':

```
\def\yes{Yes } \def\no{No } \newif\ifgarbage
\loop\message{Are you happy? }
  \read-1 to\answer
  \ifx\answer\yes\garbagefalse % the answer is Yes
    \else\ifx\answer\no\garbagefalse % the answer is No
      \else\garbagetrue\fi\fi % the answer is garbage
  \ifgarbage\message{(Please type Yes or No.)}
\repeat
```

▶ **练习20.20**
用 \loop...\repeat 原理了编写一个一般的 \punishment 宏, 它重复任意给定段落任意给定次。例如,

```
\punishment{I must not talk in class.}{100}
```

将得到练习 20.1 所要的结果。

前 30 个素数为 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, and 113。你可能觉得这没什么; 但是你可能吃惊的是上面这句话由下面所排版出:

```
前~30~个素数为~\primes{30}。
```

TEX 通过展开宏 \primes 执行了所有的计算, 因此作者可以确信上面所给素数列没什么输入错误。下面就是此宏集:

```
\newif\ifprime \newif\ifunknown % boolean variables
\newcount\n \newcount\p \newcount\d \newcount\a % integer variables
\def\primes#1{2,~3% assume that #1 is at least 3
  \n=#1 \advance\n by-2 % n more to go
  \p=5 % odd primes starting with p
  \loop\ifnum\n>0 \printifprime\advance\p by2 \repeat}
\def\printp{, % we will invoke \printp if p is prime
  \ifnum\n=1 and~\fi % ‘and’ precedes the last value
  \number\p \advance\n by -1 }
\def\printifprime{\testprimality \ifprime\printp\fi}
\def\testprimality{{\d=3 \global\primetrue
  \loop\trialdivision \ifunknown\advance\d by2 \repeat}}
\def\trialdivision{\a=\p \divide\a by\d
  \ifnum\a>\d \unknowntrue\else\unknownfalse\fi
  \multiply\a by\d
  \ifnum\a=\p \global\primefalse\unknownfalse\fi}
```

这种计算相当直接, 除了它有一个循环在另一个循环中; 因此, \testprimality 引入了一个额外的大括号组, 来保证内部循环控制不受外部循环的影响。当 \ifprime 被设定为真或假时, 大括号要求编写使用‘\global’。 TEX 处理此句所花费的时间比处理一个页面还要多; 宏 \trialdivision 被展开了 132 次。

做这些漂亮工作的宏 \loop 实际上非常简单。它把假定要重复的代码放在叫做 \body 的控制系列中, 并且接着用另一个控制系列重复到条件为假:

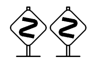
```
\def\loop#1\repeat{\def\body{#1}\iterate}
\def\iterate{\body\let\next=\iterate\else\let\next=\relax\fi\next}
```

\iterate 的展开的结尾是 \next 的展开; 因此 TEX 在调用 \next 之前能把 \iterate 从内存中清除掉, 这样在长循环中内存就不会被塞满。计算机科学家称其为“末端回归”。

下面的宏 \hex 把计数寄存器 \n 转换为十六进制表示, 它举例说明了一个递归控制结构, 许多 \hex 的副本都可以在其中同时使用。在这个应用程序中, 递归比简单的 \loop 重复更好, 因为十

六进制数字是从右到左算出来的, 而它们在输出时是从左到右。 (在 \n 中的数字必须 ≥ 0。)

```
\def\hex{{\count0=\n \divide\n by16
  \ifnum\n>0 \hex\fi \count2=\n \multiply\count2 by-16
  \advance\count0 by\count2 \hexdigit}}
\def\hexdigit{\ifnum\count0<10 \number\count0
  \else\advance\count0 by-10 \advance\count0 by‘A \char\count0 \fi}
```



我们的最后一个例子是计算变量中非空记号数目的宏; 例如, '\length{argument}'展开为'8'。它还举例说明了宏另一个方面的技术。

```
\def\length#1{{\count0=0 \getlength#1\end \number\count0}}
\def\getlength#1{\ifx#1\end \let\next=\relax
  \else\advance\count0 by1 \let\next=\getlength\fi \next}
```

21. 盒子制作

在第十一和第十二章, 我们讨论了盒子和粘连的法则, 并且现在我们已经看到这些概念的很多应用。在大多数情况下, 用 TeX 自动生成的盒子来断行, 分页和处理数学公式就可以了; 但是如果要满足特殊的要求, 就得自己来制作盒子。例如, 第十四章指出, 如果把某些内容放在 `\hbox` 中, 它就可以不被连字化或裂分; 第十九章指出, 在列表方程中可以用 `\hbox` 来得到普通文本。

本章的目的在于还未讨论的盒子的其它细节。幸运的是, 没有太多的东西要讨论; 我们已经讨论过大多数规则了, 因此本章相当短。实际上, 前面的章已经把除了标尺的规则外的几乎所有东西都讨论了。

为了制作一个标尺盒子(即, 全黑四方形), 可以在垂直模式下输入'`\hrule`'或者在水平模式下输入'`\vrule`', 其后跟部分或全部属性'`width`⟨dimen⟩', '`height`⟨dimen⟩', '`depth`⟨dimen⟩', 它们可以用任意次序。例如, 如果

```
\vrule height4pt width3pt depth2pt
```

出现在段落中间, TeX 就输出黑盒子'▮'。如果一个尺寸给出了两次, 那么第二次就冲掉了第一次的值。如果某个尺寸没有给定, 得到的就是下列默认值:

	`\hrule`	`\vrule`
width	*	0.4 pt
height	0.4 pt	*
depth	0.0 pt	*

这里'*'的意思是实际尺寸与上下文有关; 标尺要延伸到最小的盒子边界, 或者与封装它的盒子对齐。

例如, 在输入本段之前作者输入了'`\hrule`', 并且你可以看到所出现的东西: 一条水平线, 厚度为, 0.4 pt, 横贯页面, 这是因为封装它的垂直盒子就是这么宽。(实际上, 封装它的盒子是页面自己。) 另一个例子在本段落紧后面, 在那里你可以看到

```
\hrule width5cm height1pt \vskip1pt \hrule width6cm
```

的结果。在垂直列中, TeX 不在标尺盒子和它们的邻近的行之间插入行间粘连, 因此这两个标尺正好分开 1 pt。

▶ **练习21.1**

笨笨要插入一个水平标尺, 但是不希望它挨着左页边, 因此就把它放在一个盒子里, 并且把盒子向右移, 如下:

```
\moveright 1in \vbox{\hrule width3in}
```

但是他发现, 这样得到的结果, 与不用 `\vbox` 而直接用'`\hrule width 4in`'得到的标尺, 在上下都有更多的间距。为什么 TeX 插入了更多的间距, 怎样做才能避免?

如果给出了一个标尺的所有三个尺寸, 那么在 \hrule 和 \vrule 之间没有什么本质的不同, 因为它们得到完全一样的黑盒子。但是要把它放在垂直列中就必须用 \hrule, 放在水平列中必须用 \vrule, 而不管它看起来象一个水平标尺, 还是垂直标尺, 或者都不象。如果在垂直模式下输入 \vrule, TEX 就另起一段; 如果在水平模式下输入 \hrule, TEX 就停止当前段落并且返回垂直模式。

标尺的尺寸可以是负值; 例如, 后面的标尺高度为 3 pt, 深度为 −2 pt: ‘———’。但是, 只有高度加深度为正并且宽度为正时, 此标尺才是可见的。宽度为负的标尺不可见, 但是当出现在水平列中时, 它实际上就象后退一格一样。

▶ **练习21.2**
看看作者怎样才能得到前一段的标尺‘———’。[提示: 其长度为一英寸。]

现在我们总结一下 TEX 中所有直接给出盒子的方法。(1). 在水平模式下, 字符自己制作出字符盒子; 此字符是从当前字体中取得的。(2). 命令 \hrule 和 \vrule 制作出标尺盒子, 就象刚刚讨论的那样。(3). 另外, 你可以自己制作 hbox 和 vbox, 它们都属于一般条件 ⟨box⟩ 的一种。⟨box⟩ 有下面七种形式:

\hbox⟨box specification⟩{⟨horizontal material⟩}	(见第 12 章)
\vbox⟨box specification⟩{⟨vertical material⟩}	(见第 12 章)
\vtop⟨box specification⟩{⟨vertical material⟩}	(见第 12 章)
\box⟨register number⟩	(见第 15 章)
\copy⟨register number⟩	(见第 15 章)
\vsplit⟨register number⟩to⟨dimen⟩	(见第 15 章)
\lastbox	(见第 21 章)

在这里, ⟨box specification⟩ 为‘to⟨dimen⟩’, ‘spread⟨dimen⟩’或是空的; 它控制着盒子中水平或垂直列中的粘连是多少, 就象第十二章讨论的那样。⟨register number⟩ 在 0 和 255 之间; 在使用 \box 之后, 此寄存器被置空, 但是使用 \copy 后, 寄存器不变, 就象第十五章讨论的那样。命令 \vsplit 也在第十五章讨论过了。在数学模式下, 还可以用另外一种盒子: \vcenter⟨box specification⟩{⟨vertical material⟩} (见第十七章)。

上面表中的最后一行给出了 \lastbox, 它是一个以前没有讨论过的原始命令。如果在当前水平列或垂直列中的最后一个项目是一个 hbox 或 vbox, 就把它从此列中删去, 并且把它变成 \lastbox; 否则, \lastbox 是置空的。这个命令允许出现在内部垂直模式, 水平模式和受限水平模式, 但是不能用它代替垂直模式下来自当前页面的盒子。在数学模式下, \lastbox 总是置空的。在段落的开头, ‘{\setbox0=\lastbox}’就把段落缩进的盒子去掉了。

命令 \unskip 基本上同 \lastbox 一样, 只是它应用在粘连上而不是盒子上。如果当前列上的最后一项是一个粘连项目(或者引导符, 下面要讨论它), 就去掉它。在垂直模式下, 不能用 \unskip 从当前页面去掉粘连, 但是可以用‘\vskip-\lastskip’, 这样得到的效果基本一样。

第二十四至二十六章总结了所有模式下 TEX 的所有命令, 并且这些总结中所提到的'⟨box⟩'都是刚才列表中七种的某个。例如, 在任何模式下, 都可以使用'`\setbox`⟨register number⟩`=`⟨box⟩', 并且可以在垂直模式下使用'`\moveright`⟨dimen⟩⟨box⟩'。但是要使用'`\setbox`⟨register number⟩`=C`' 或'`\moveright`⟨dimen⟩`\hrule`'却不行; 那样 TEX 会抱怨说 ⟨box⟩ 应该已存在才行。字符的盒子太特殊, 不被看作 ⟨box⟩。

▶ **练习21.3**
定义一个控制系列 `\boxit`, 使得'`\boxit{`⟨box⟩`}`'把所给定的盒子在四面用双线条(线条间距为 3 pt)框起来。

> 例如, 现在你看到的语句就是列表公式 `$$\boxit{\boxit{\box4}}$$` 的一部分, 其中 `\box4` 是这样生成的: '`\setbox4=\vbox{\hsize 23pc \noindent \strut 例如, 现在你看到的语句 ... \strut}`'。

现在我们再看看盒子中可以放什么。一个 hbox 包含一个水平列; 一个 vbox 包含一个垂直列。两种列最终都是由象盒子, 粘连, kern 和 penalty 这样的东西组成, 就象我们已经在第十四和十五章讨论过的那样。但是, 还可以包括某些我们还未讨论过的特殊的东西, 即"引导符"和"无名"。在本章剩下的部分, 我们就看看怎样来利用这些奇特的项目。

在你眼前的一系列圆点 . . . . 被称为引导符因为它们把你的视线从页面一边引导到另一边; 这样的东西经常出现在索引和目录中。一般思路是把一个盒子重复到填满所给定的空间这么多次。 TEX 把引导符看作一种特殊的粘连; 别, 等一下, 还有另一种说法: TEX 把粘连看作一种特殊的引导符。普通粘连用空白充满空间, 而引导符是用任何想用的东西充满空间。在水平模式下, 可以使用

`\leaders`⟨box or rule⟩`\hskip`⟨glue⟩

并且它的效果与只使用'`\hskip`⟨glue⟩'一样, 除了在空间中被重复的是给定的 ⟨box or rule⟩ 之外。粘连的伸缩与平常一样。例如,

```
\def\leaderfill{\leaders\hbox to 1em{\hss.\hss}\hfill}
\line{Alpha\leaderfill Omega}
\line{The Beginning\leaderfill The Ending}
```

将得到下面两行:

Alpha . . . . Omega
The Beginning . . . . The Ending

这里的'`\hbox to 1em{\hss.\hss}`'给出了宽度为一个 em 的盒子, 句点在盒子的中央; 因此当在盒子 `\line` 要充满空间时, 控制系列 `\leaderfill` 就重复此盒子。(Plain TEX 的宏 `\line` 生成的是一个宽度为 `\hsize` 的 hbox。)

注意: 在这两个例子中, 看起来圆点上下正好对齐了。这不是巧合; 其原因是, 命令 `\leaders` 的作用象一个窗格一样, 你所看到的只是无限格盒子的一部分。如果单词'Alpha'和'Omega'被长的单词代替, 圆点的数目会少, 但是能看到的圆点的位置却不变。无限重复的盒子排列起来使得它们互相对齐, 并且如果它们都能被可见的话, 它们之一作为最小的封装盒子都有同样的参考点。因此, `\leaders` 把盒子按所封装盒子的左边界对齐, 如果从那里开始引导符的话; 但是不能得到右对齐的盒子, 除非封装盒子的宽度与重复后盒子的宽度正好整除。如果重复后盒子的宽度为 w, 并且要充满的空间至少为 $2w$, 那么总是可以至少看见一个盒子; 但是如果空间小于 $2w$, 盒子可能显不出来, 因为在无限列中, 只有当盒子的整个宽度完全落在可用空间中时才把盒子排版出来。

当引导符互相独立时, 你可能不希望它们象刚才讨论的那样互相对齐, 因此, TEX 还提供了非对齐的引导符。在这种情况下, 当要被充满空间大于 qw 且小于 $(q+1)w$ 时, 宽度为 w 的盒子就重复 q 次; 还有, 所得到的结果在此空间中将居中。在 TEX 中有两种非对齐的引导符, 即 `\cleaders` (居中引导符)和 `\xleaders` (伸开的引导符)。居中的引导符把盒子紧挨着排列, 在左右两头留出相等的空白; 伸开的引导符把空白平均放在 q 个盒子两边的 $q+1$ 个接口处。例如, 假设用于引导符的盒子为 10 pt 宽, 要充满 56 pt 的空间。因此要用五次盒子; `\cleaders` 得到的是 3 pt 的空白, 接着是五个盒子, 再接下来是 3 pt 的空白。但是 `\xleaders` 得到的是 1 pt 的空白, 接着是第一个盒子, 接下来又是 1 pt 空白, 再接下来是第二个盒子, ..., 接着是第五个盒子和 1 pt 的空白。

▶ **练习21.4**

假设有一个 10 pt 宽的盒子, 要充满的是 38 pt 的空间, 此空间从封装它的盒子左边界的 91 pt 处开始。那么, 用 `\leaders`, `\cleaders` 和 `\xleaders` 各得到多少个盒子? 在每种情况下, 相对于封装盒子的左边界, 盒子被放在什么地方?

▶ **练习21.5**

假定在前面的宏 `\leaderfill` 中, '`.`'只有 0.2 em 宽, 在一个 em 的盒子中, 两边各有 0.4 em 的空白。因此, `\leaders` 所得到的结果中, 句点和两端文本之间的空白在 0.4 em 到 1.4 em 之间。重新定义 `\leaderfill`, 使得在两端的空白在 0.1 em 到 1.1 em 之间, 但是相邻的引导符行仍然要对齐。

在引导符的构造中, ⟨box⟩ 可用 ⟨rule⟩ 来代替, 它可用 `\hrule` 或 `\vrule`, 后面象通常那样可跟着 `height`, `width` 和 `depth` 这些量。这样得到的标尺与相应的 ⟨glue⟩ 一样宽。在此, `\hrule` 在水平模式下是可用的, 因为它给出的是文本中的水平标尺。例如, 如果把我们前面例子中的宏 `\leaderfill` 变成

```
\def\leaderfill{ \leaders\hrule\hfill\ }
```

那么所得到的结果就是

Alpha ________________________________ Omega
The Beginning ________________________ The Ending

当用标尺代替盒子时, 它要把空间全部充满, 因此 `\leaders`, `\cleaders` 和 `\xleaders` 之间没有差别。

▶ **练习21.6**

`\leaders\vrule\hfill` 得到的是什么?

引导符也可以用在垂直模式下, 就象在水平模式下一样。在这种情况下, 垂直粘连(比如, \vskip⟨glue⟩ 或 \vfill)用来代替水平粘连, 并且 \leaders 得到的盒子是这样对齐的: 每个所重复盒子的顶部与最小封装盒子的顶部处于同一垂直地点, 再加上若干个所重复盒子的深度加高度。在垂直引导符中, 不插入行间粘连; 盒子正好互相堆在顶部。

如果所给水平引导符的盒子宽度不是正值, 或者垂直引导符的盒子的深度加高度不是正值, 那么 TEX 就暗中忽略掉此引导符, 并且转而生成一个普通的粘连。

▶ **练习21.7**
看看怎样用一个标尺来结束一个段落, 此标尺最少为 10 pt 长, 把在任何情况下都要伸到右边界, 象这样: ____

水平引导符与水平粘连略有不同, 因为当计算封装盒子的尺寸时, 水平引导符有高度和深度(即使它们使用的次数为零)。类似地, 垂直引导符也有宽度。

▶ **练习21.8**
通过在水平引导符中使用垂直引导符, 看看怎样得到下列'TEX 网':

TEXTEXTEXTEXTEXTEXTEXTEXTEXTEX
TEXTEXTEXTEXTEXTEXTEXTEXTEXTEX
TEXTEXTEXTEXTEXTEXTEXTEXTEXTEX
TEXTEXTEXTEXTEXTEXTEXTEXTEXTEX

(把 TEX 标志放在四方盒子中, 再紧堆在一起。)

▶ **练习21.9**
用垂直引导符解出练习 20.1。

Plain TEX 的宏 \overbrace 和 \underbrace 可以通过把字符和标尺组合起来而得到。字体 cmex10 中有四个符号 ⌣⌢⌢⌣, 它们的深度为零, 高度正好等于连接它们的标尺的厚度。因此, 可以很容易给出 \upbracefill 和 \downbracefill 的定义, 从而在垂直模式下通过'\hbox to 100pt{\downbracefill} \hbox to 50pt{\upbracefill}'得到, 比如,

$\overbrace{\hspace{100pt}}$
$\underbrace{\hspace{50pt}}$

这些宏的详细定义见附录 B。

附录 B 中 \overrightarrow 的定义比 \overbrace 更复杂, 因为它用到的是盒子而不是标尺。所设计的 plain TEX 字体要使得象 ← 和 → 这样的符号可以用减号来加长; 类似地, ⇐ 和 ⇒ 要用等号来加长。但是, 不能把它们简单地依次放在一起, 因为那样会留下空隙('←−−−'和'⇐=='); 必须在字符之间退回一点点。更复杂的情况出现在加长的箭头所要的长度不是减号长度的整数倍。为了解决这个问题, 附录 B 中的宏 \rightarrowfill 使用了 \cleaders 和一个要重复的盒子, 此盒子由减号的中间 10 个单位组成, 其中一个单位是 $\frac{1}{18}$ em。引导符放在前面, 接着是 − 和 →; 所退回的长度足以填补前后大约 5 个单位的额外空间, 这样 \cleaders 就不会留下空白了。用这种方法, 下列宏

```
\hbox to 100pt{\rightarrowfill}
```

得到的就是'—————————→'。

现在我们知道了引导符。那么什么是无名呢? 嗯, 无名是作为一个一般方法而给出的, 通过它可以把 TEX 扩展到应用于处理重要的特殊输出。它使得系统高手可以修改 TEX 程序, 而不用改变太多代码, 这样可以很快适应新的方法, 而不是编写宏。作者希望不要经常进行这样的扩展, 因为所要的不是越来越不兼容的赝 TEX 系统; 他还认识到, 某些特殊的方面应当特殊对待。无名使得可以把新东西加入到盒子中而不需要太多地改变已有的约定。但是它们使得应用程序更难从一台机器移植到另一台上。

作为整个 TEX 系统的一部分, 定义了两种无名。它们其实不是 TEX 的扩展, 但是就得这样分类, 这样就为可能出现的扩展提供了一个模型。第一个与输出到文本文件有关, 它包括 TEX 原始命令 `\openout`, `\closeout`, `\write` 和 `\immediate`。第二个与特殊指令有关, 它可以把特殊指令通过 TEX 的命令 `\special` 传送给输出设备。

写入所得到的文本文件可以在以后作为其它程序(包括 TEX)的输入文件, 因此这个功能使得 TEX 可以完成目录, 索引和其它别的工作。象第二十章的命令 `\openin` 和 `\closein` 那样, 可以用的命令有'`\openout`⟨number⟩`=`⟨file name⟩'和'`\closeout`⟨number⟩'; ⟨number⟩ 必须在 0 和 15 之间。如果文件名没有扩展名, 通常就添加上'`.tex`'。有一个 `\write` 命令, 它把一行写入文件中, 类似于命令 `\read` 读入一行; 使用

`\write`⟨number⟩`{`⟨token list⟩`}`

并且内容就输出到对应于给定输出流编号的文件中。如果 ⟨number⟩ 为负值或者大于 15, 或者所给出的输出流没有为输出而打开的文件, 那么就输出到用户的 log 文件中, 如果编号不是负值, 还输出到终端上。Plain TEX 用命令 `\newwrite` 来分配从 0 到 15 的输出流编号。输出流完全独立于输入流。

然而, 输出实际上是以延迟的方式进行的; 在 TEX 遇见你给出的 `\openout`, `\closeout` 和 `\write` 命令时, 它并不执行。而是 TEX 把这些命令放在无名项目中, 并且把它们放在正在构建的水平列, 垂直列或数学列中。直到最后这些无名作为一个更大盒子的一部分而被送到 `dvi` 文件时, 才有实际的输出。这种延迟的原因是, `\write` 经常用在制作索引和目录, 而在 `\write` 指令出现在段落中间的时候, 一般并不知道要在哪个页面放这些特殊的项目。 TEX 通常超前运行, 在断行之前要读入整个段落, 在分页之前要收集的行比需要的要多, 就象在第十四和十五章中讨论的那样。因此, 为了确保所引用的页码的准确性, 唯一的方法是采用延迟写入机制。

命令 `\write` 的 ⟨token list⟩ 首先存储在一个无名项目中, 而不进行宏展开; 稍后, 当 TEX 处在 `\shipout` 执行过程时, 再进行宏展开。例如, 假定在文稿中的某些段落中间有以下文本:

```
... For \write\inx{example: \the\count0}example, suppose ...
```

那么段落的水平列将在单词'`example`'紧前面, '`For`'后面空格紧后面有一个无名项目。这个无名项目包含未展开的记号列'`example: \the\count0`'。当段落断行并且放在当前页面时, 它处在休眠状态。让我们假设此单词'`example`'(或者它的某些连字的前半部分, 象'`ex-`')被输送到第 256 页。那么 TEX 将把行

```
example: 256
```

写入到输出流 \inx, 因为此时'\the\count0'将被展开。当然, 命令 \write 通常由宏来调用; 它们很少直接出现在段落中间。

TEX 也通过把命令 \openout 和 \closeout 放在无名项目中而延迟它们; 因此, 输出命令的相对顺序将保留下来, 除非由于插入或其它此类东西把盒子按其它次序输送出去了。

有时候不希望 TEX 延迟 \write, \openout 或者 \closeout。那么可以用, 比如, '\shipout\hbox{\write...}', 但是它将会在你的 dvi 文件中产生一个不想要的空页。因此, TEX 用另一种方法来绕过这个问题: 如果在 \write, \openout 或 \closeout 的紧前面加上'\immediate', 那么命令将立即被执行, 并且不生成无名项目。例如,

```
\immediate\write16{Goodbye}
```

在你的终端上输出'Goodbye'。在没有 \immediate 时, 在当前列输出之前, 你不会看到'Goodbye'。(实际上, 可能你永远看不到它; 或者不止一次看到它, 只要当前列被送到被拷贝的盒子中就是这样。) '\immediate\write16'与 \message 不同, \write 把文本单独放在一行, 而几个 \message 命令得到的结果出现在同一行, 中间用空格隔开。

\write 的 ⟨token list⟩ 应当相当短, 因为它只输出一行。TEX 的某些命令不能输出长的行; 如果要输出很多东西, 就用几个 \write 命令。另外, 也可以把 TEX 的 \newlinechar 参数设置为某些字符的 ASCII 代码数, 用它们来表示"回车换行"; 这样, 只要它正常把此字符输出到文件中, TEX 就开始一个新行。例如, 在一个 \write 命令中输出两行, 可以用

```
\newlinechar=`\^^J
\immediate\write16{Two^^Jlines.}
```

每个 \write 命令生成的输出总是采用 TEX 用符号显示记号列的方法: 字符得到的是它自己(除了对宏参数字符 ## 得到的是两个字符); 不能展开的控制系列得到的是其名字, 其前面是 \escapechar, 后面是一个空格(除非名字是活动符, 或者控制系列是单个非字母)。

TEX 将忽略掉出现在由引导符控制的盒子中的无名项目 \write, \openout 和 \closeout。要想违反是不行的。

因为延迟 \write 的 ⟨token list⟩ 在相当随意的时间进行展开(当执行 \shipout 时), 因此对允许包含它的控制系列要小心。第二十章的宏展开的方法使处理 \write 更容易。

▶ **练习21.10**
假设要 \write 一个记号列, 它由一个包含当前章节编号的宏 \chapno, 以及对应于当前页的'\the\count0'组成。而你希望 \chapno 立即展开, 因为在记号列输出之前它可能会改变; 但是 \the\count0 要在 \shipout 时展开。怎样编写它?

现在, 我们回来再讨论盒子, 它还有一个特征。命令'\special{⟨token list⟩}'可以在任何模式下使用。象 \write 一样, 它也把记号列放在无名项目中; 并且象 \message 一样把记号列立即展开。这个记号列与 TEX 产生的其它排版命令一起输出到 \dvi 文件中。因此, 它暗中对应于页面的一个特殊

位置, 即就象代替无名项目的是一个高度, 深度和宽度都是零的盒子时的基准点。在 `\special` 命令中的 ⟨token list⟩ 由关键词组成, 在必要时要跟上一个空格或相应的变量。例如,

```
\special{halftone pic1}
```

就意味着文件 `pic1` 中的图像要插入到当前页面, 其基准点在当前位置。 TEX 不理会记号列的意思; 只把它拷贝到输出中。但是, 不要用过长的记号列, 否则 TEX 的字符串内存可能会溢出。可以把命令 `\special` 用在可用的特殊设备上, 比如, 用来漂亮的颜色了打印书籍。

把 `dvi` 文件转换为打印或显示输出的软件可能因为不认识你的特殊关键词而转换失败。因此, 命令 `\special` 不能完成改变当前位置的工作。只要用到 `\special`, 就意味着你的输出文件并不能在所有输出设备上打印, 因为所有的 `\special` 函数都是 TEX 的扩展。但是, 作者希望在 TEX 用户协会中形成某些公共图形命令的标准, 它可以由各方面的人通过详细的实验后制订出来; 这样, 在使用 `\special` 扩展时就可以有某些规则了。

在制作盒子后可得到 `\badness` 的数值, TEX 会给出设置在盒子中粘连的 badness。例如, 可以使用

```
\setbox0=\line{\trialtexta}
\ifnum\badness>250 \setbox0=\line{\trialtextb}\fi
```

badness 在 0 和 10000 之间, 如果盒子溢出了, 那么 `\badness=1000000`。

22. 对齐方式

当你要求印刷工人排版表格时, 他们要进行特殊处理, 而且这样做也有充分的理由: 每个表格都有各自的特点, 因此必须对每个都要有些侧重, 并且在几种方法中互相比较以得到漂亮而清晰的结果。但是, 用 TeX 处理就不需要太担心, 因为 plain TeX 有一个"制表"命令, 即使在打字机上你也可以把简单的表格变成漂亮的格式。还有, TeX 有一个强大的对齐方法, 这使得它能够处理非常复杂的列表格式。这些对齐命令的简单运用就足以完成大部分工作了。

我们首先来讨论制表。如果给出'\settabs n \columns', plain TeX 就生成了等分为 n 栏的行。每行的内容用下列方法输入:

\+$\langle\mathrm{text}_1\rangle$&$\langle\mathrm{text}_2\rangle$&$\cdots$\cr

其中 $\langle\mathrm{text}_1\rangle$ 左页边对齐, $\langle\mathrm{text}_2\rangle$ 从第二栏的左边界开始, 等等。注意, 行开头是'\+'。最后一栏要跟'\cr', 它就是以前的"回车"命令。例如, 看看下面的例子:

```
\settabs 4 \columns
\+&&此文本在第三栏\cr
\+&此文本在第二栏\cr
\+\KT{10}此文本在第一栏和&&&
  第四栏以及&超出页面!\cr
```

在'\settabs4\columns'之后, 每个 \+ 行被四等分, 因此得到的结果是

		此文本从第三栏开始		
	此文本从第二栏开始			
此文本从第一栏开始和			第四栏以及	超出页面!

要好好研究一下这个例子, 因为它说明了好几个问题。(1). '&'象许多打字机的TAB键一样; 它指示 TeX 缩进到下一个制表符的位置, 在每栏的右边界有一个制表符。本例中, TeX 设置了四个制表符, 如图用虚线来表示它们; 虚线还出现在左页边, 其实它不是一个制表符。(2). 但是'&'与机械打字机的TAB不完全一样, 因为在缩进到下一栏前, 它首先要回退到当前栏的开头。这样, 通过计算 & 的数目, 就可以知道正在制表的是哪一栏; 这很方便, 因为如果宽度会变化, 就很难知道你是否超过了制表符的位置。因此, 在本例中的最后一行, 为了得到第四栏, 在前面输入了三个 &, 而不管文本是否已经超出第一栏甚至到了第三栏。(3). 如果本行已经输入完毕, 不管是否后面还有空栏, 都可以输入'\cr'来结束本行。(4). & 与制表符还有一个方面不同: TeX 忽略掉'&'后面的空格, 因此, 在输入完一栏时, 可以把'&'放在行尾, 而不会出现额外的空格。(本例中的倒数第二行以'&'结尾, 而且在此符号后面暗中跟着一个空格; 如果 TeX 没有忽略掉此空格, 那么文本"第四栏"就不会正好出现在第四栏的开头了。) 顺便说一下, plain TeX 还忽略掉'\+'后面的空格, 这样第一栏就与其它栏一样了。(5). 在例子的最后一行中, 虽然没有用大括号来界定楷体作用的范围, 但是'\KT{10}'只把第一栏变成楷体, 这是因为 TeX 在暗中在各个当前单元外围插入了大括号。

一旦声明了命令 \settabs, 制表符就一直保持到你重新设置为止, 即使你输入的是象平常那样的普通段落。但是, 如果把 \settabs 封装在 {...} 中, 定义在组中的制表符不会影响外面的制表符; 不允许使用'\global\settabs'。

制表的行通常用在段落之间, 与使用 \line 或 \centerline 的位置一样, 它们得到也是特殊形式的行。但是, 把 \+ 的行放在 \vbox 也是有用处的; 用它可以很方便地给出包含对齐内容的列表公式。例如, 如果给出

```
$$\vbox{\settabs 3 \columns
  \+这是&列表方程&三栏\cr
  \+格式的&一个奇怪&例子。\cr}$$
```

你所得到的是下列列表公式:

这是	列表方程	三栏
格式的	一个奇怪	例子。

在这种情况下, 第一栏没有居左, 这是因为 TEX 把列表公式的盒子居中了。在 \+ 的行中, 要把以 \cr 结尾的栏中内容放在其自然宽度的盒子中; 因此, 这里的第一和第二栏的宽度是 \hsize 的三分之一, 但是第三栏的宽度为文本"例子。"的宽度。在这个构造中, 即使没有任何数学内容, 我们也用了 \$\$, 这是因为 \$\$ 自有其用处; 例如, 它把盒子居中, 并且在上下插入了间距。

人们还希望非等分的制表符, 因此我们提供了另一种方法: 在'\settabs'紧接着给出'\+⟨sample line⟩\cr'。在这种情况下, 制表符被放在例句中 & 的位置上, 并且例句自己并不出现在输出中。例如,

```
\settabs\+\indent&Horizontal lists\quad&\cr % sample line
\+&Horizontal lists&Chapter 14\cr
\+&Vertical lists&Chapter 15\cr
\+&Math lists&Chapter 17\cr
```

排版出下列三行内容:

Horizontal lists	Chapter 14
Vertical lists	Chapter 15
Math lists	Chapter 17

在本例中, 命令 \settabs 使第一栏宽度与段落的缩进一样宽; 第二栏与'Horizontal lists'加上一个 quad 间距的宽度一样。在本例中只设定了两个栏的宽度, 因为在例句中只出现了两个 &。(例句也可以用 & 来结尾, 因为跟着最后一个制表符的文本没起什么作用。)

表格的第一行并不能总是作为例句使用, 因为它有时候不能给出正确的制表符位置。在一个大表格中, 你必须浏览一下, 找出每个栏中最大的单元; 这样, 例句就由最宽的第一栏, 最宽的第二栏, 等等, 忽略掉最后一栏这样来构造。在例句中要确保有某些额外的间距, 这样栏才不会互相紧挨着。

▶ **练习22.1**

看看怎样排版下列表格 [from Beck, Bertholle, and Child, *Mastering the Art of French Cooking* (New York: Knopf, 1961)]:

Weight	*Servings*	*Approximate Cooking Time**
8 lbs.	6	1 hour and 50 to 55 minutes
9 lbs.	7 to 8	About 2 hours
9 1/2 lbs.	8 to 9	2 hours and 10 to 15 minutes
10 1/2 lbs.	9 to 10	2 hours and 15 to 20 minutes

* For a stuffed goose, add 20 to 40 minutes to the times given.

如果要把某些内容在栏中居右, 就在其前面使用'`\hfill`'; 并且在其后要有'`&`', 这样就确保了 TEX 排版的是下一个制表符前面的所有内容。类似地, 要把某些内容在栏中居中, 在前面使用'`\hfill`', 在后面使用'`\hfill&`'。例如,

```
\settabs 2 \columns
\+\hfill 此内容在前半行&
    \hfill 此内容在后半行\hfill&\cr
\+\hfill 居右。&
    \hfill 居中。\hfill&\cr
```

得到的是下列小表格:

此内容在前半行	此内容在后半行
居右。	居中。

附录 B 中的宏 `\+` 是把跟在 `&` 后每栏的 ⟨text⟩ 放在如下的一个 hbox 中:

```
\hbox to ⟨column width⟩{⟨text⟩\hss}
```

`\hss` 就意味着文本在正常情况下是居左的, 并且它可以延伸到盒子的右边。因为 `\hfill` 比 `\hss` 的伸缩能力更强, 所以可以得到上面的居右或居中的效果。注意, `\hfill` 不收缩, 但是 `\hss` 收缩; 如果文本在栏中放不下, 它将在右边伸出来。通过添加 `\hfilneg` 可以取消 `\hss` 的收缩性; 这样, 超过尺寸的文本得到的就是一个溢出的盒子。你还可以把'`\hss`'放在此内容前面或其后'`&`'紧前面来把它居中; 在这种情况下, 文本允许从左右两边伸出栏。`\+` 行的最后一栏(即, 跟着 `\cr` 的栏单元)的处理方法不同: 只把文本放在其自然宽度的 hbox 中。

计算机程序出现的是另一种不同的困难, 因为在一种格式中需要行和行之间的制表符位置不断变化。例如, 看看下列程序片段:

if $n < r$ **then** $n := n + 1$
  else begin *print_totals*; $n := 0$;
    end;
while $p > 0$ **do**
  begin $q := link(p)$; *free_node*(p); $p := q$;
  end;

要设置特殊的制表符使得'**then**'出现在'**else**'上面, 并且'**begin**'出现在'**end**'上面。只要需要新的制表符位置, 就可以通过设置新的例句来得到它; 但是这是一项无聊的工作, 因此 plain TEX 给出了一种简单方法。只要在已经有的制表符右边给出 &, 就在那里设定了一个新的制表符, 用这种方法, 栏都处在自然宽度的盒子中。还有, 用命令'`\cleartabs`'可以重新设置当前栏右边的所有制表符的位置。因此, 上面的计算机程序可以如下用 TEX 排版:

```
$$\vbox{\+\bf if $n<r$ \cleartabs&\bf then $n:=n+1$\cr
  \+&\bf else &{\bf begin} ${\it print\_totals}$; $n:=0$;\cr
  \+&&{\bf end};\cr
  ⟨剩下的作为练习⟩}$$
```

▶ **练习22.2**
完成上面例子中的计算机程序。

虽然行 `\+` 可以用在垂直盒子中, 但是不要在 `\+` 行中使用另一个 `\+`。宏 `\+` 只能单个使用。

附录 B 中的宏 `\+` 和 `\settabs` 是这样给出制表符的: 把寄存器 `\box\tabs` 作为一个盒子使用, 此盒子由各个栏这么宽的空盒子按照逆序充满。因此, 用'`\showbox\tabs`'可以检验当前设置的制表符; 它把栏宽度按照从右到左的顺序显示在你的 log 文件中。例如, 在给出'`\settabs\+\hskip100pt&\hskip200pt&\cr\showbox\tabs`'后, TEX 将显示出下列行:

```
\hbox(0.0+0.0)x300.0
.\hbox(0.0+0.0)x200.0
.\hbox(0.0+0.0)x100.0
```

▶ **练习22.3**
研究一下附录 B 中的宏 `\+`, 看看怎样让它象打字机的制表符那样使用(即, 使得'`&`'总是移动到当前位置右边的下一个制表符处)。假设用户不能回退到前一个制表符的位置; 例如, 如果输入为'`\+&&\hskip-2em&x\cr`', 别管第一和第二栏, 只把'x'放在第三栏的开头。(这个练习有点难。)

TEX 还有制表的另一种重要方法, 就是使用命令 `\halign`("水平对齐")。在这种情况下, 表格的样式建立在模板的概念上, 而不是制表; 思路是在每栏中给出一个单独的文本环境。各个单元插入到其模板中, 这样很快就制作出表格了。


例如, 再回头看看在本章前面出现过的水平/垂直/数学列的例子; 我们可以不用制表符, 而用 `\halign` 来得到它。新方法是

```
\halign{\indent#\hfil&\quad#\hfil\cr
  Horizontal lists&Chapter 14\cr
  Vertical lists&Chapter 15\cr
  Math lists&Chapter 17\cr}
```

并且它得到的结果与原来的一样。这个例子值得仔细研究, 因为一旦你掌握了 `\halign` 的窍门, 它实际上就相当简单了。第一行包含了对齐的导言, 它有点象设置 `\+` 的制表符例句。在本例中, 导言包含两个模板, 即第一栏的'`\indent#\hfil`'和第二栏的'`\quad#\hfil`'。每个模板正好出现一次'`#`', 它的意思是"每个栏单元的文本放在此处"。因此, 当用'`Horizontal lists`'来填充其模板时, 导言后面的行的第一栏就变成

```
\indent Horizontal lists\hfil
```

类似地, 第二栏变成'`\quad Chapter 14\hfil`'。问题是, 为什么要用 `\hfil`? 噢, 现在我们渐入佳境了: TeX 在排版之前首先把整个 `\halign{...}` 的内容读入其内存, 并且确定每栏的最大宽度, 假定每栏都没有设置伸缩的粘连。这时, 它才回过头来把每个单元放在一个盒子中, 并且设定粘连使得每个盒子的栏宽度都是最大的。这就是为什么要用到 `\hfil`; 它在窄单元中伸长以充满额外的空间。


▶ **练习22.4**

如果在本例第一栏的模板中用'`\indent\hfil#`'代替'`\indent#\hfil`', 得到的表格是什么样?


在进一步读下去前, 一定要理解了刚才例子中模板的思想。在 `\halign` 和 `\+` 之间有结果重要的差别: (1). `\halign` 自动计算最大栏宽度; 不必去估计最长的单元是哪个, 就象在例句中设置制表符时所做的那样。(2). 每个 `\halign` 要计算自己的栏宽度; 如果要让两个不同的 `\halign` 命令得到同样的对齐, 就必须做特殊处理。相反, 命令 `\+` 一直记着制表符的位置, 直到它们被重新设定为止; 在 `\+` 之间插入任意个段落, 甚至插入命令 `\halign` 也不影响制表符。(3). 因为 `\halign` 为了计算最大栏宽度要读入整个表格, 所以它不适宜用在书中横贯几页的大表格。相反, 命令 `\+` 一次只处理一行, 因此它对 TeX 内存中没有什么特殊要求。(但是, 如果要生成一个大表格, 可能你要为每行定义特殊的宏而不是只靠一般的 `\+` 命令。) (4). `\halign` 花的计算机时间比 `\+` 少, 因为 `\halign` 是 TeX 的内置命令, 而 `\+` 是一个宏, 它由 `\halign` 和其它原始命令一起构成。(5). 模板比制表符更通用, 并且节省很多输入工作量。例如, 水平/垂直/数学列的表格中, 如果注意到栏中的公共部分, 就可以如下更简洁地输入它:

```
\halign{\indent# lists\hfil&\quad Chapter #\cr
  Horizontal&14\cr Vertical&15\cr Math&17\cr}
```

甚至于如果注意到了章的编号以'`1`'开头, 那么还可以节省两次击键! (注意: 可能象这样优化比直接输入花费的时间更多; 只有你无聊或寻乐时再用它。) (6). 另一方面, 模板不能代替制表符, 因为制表符的位置是连续变化的, 就象在计算机程序的例子中一样。

我们用一个更有意思的表格来进一步熟悉 `\halign`。下面是另一个例子:

American Chicken	*French Connection*	*Age (months)*	*Weight (lbs.)*	*Cooking Methods*
Squab	*Poussin*	2	$^{3}/_{4}$ to 1	Broil, Grill, Roast
Broiler	*Poulet Nouveau*	2 to 3	$1\,^{1}/_{2}$ to $2\,^{1}/_{2}$	Broil, Grill, Roast
Fryer	*Poulet Reine*	3 to 5	2 to 3	Fry, Sauté, Roast
Roaster	*Poularde*	$5\,^{1}/_{2}$ to 9	Over 3	Roast, Poach, Fricassee
Fowl	*Poule de l'Année*	10 to 12	Over 3	Stew, Fricassee
Rooster	*Coq*	Over 12	Over 3	Soup stock, Forcemeat

注意, 除了标题行外, 第一栏的设置为居右, 使用 bold 字体; 中间的栏是居中的; 第二栏居中, 使用的是 italic 字体; 最后一栏是居左。我们希望尽可能简单地输入表格的行; 因此, 例如, 最好是只用输入

```
Rooster&Coq&Over 12&Over 3&Soup stock, Forcemeat\cr
```

就能得到最后一行, 而不用管字体, 居中与否等等。这不但减小输入工作量, 还能减少输入中的错误。因此, 第一栏的模板应该是'`\hfil\bf#`'; 第二栏的模板应该是'`\hfil\it#\hfil`', 得到的是居中的 italic 文本; 等等。我们还要在栏之间插入一定的间距, 比如一个 quad。

```
\halign{\hfil\bf#&\quad\hfil\it#\hfil&\quad\hfil#\hfil&
          \quad\hfil#\hfil&\quad#\hfil\cr
 ⟨the title lines⟩
 Squab&Poussin&2&\frac3/4 to 1&Broil, Grill, Roast\cr
 ... Forcemeat\cr}
```

就象 `\+` 命令一样, 在导言研究表格的各个行中, `&` 后面的空格被忽略掉。因此, 当输入文件中表格的行要占多行时, 用'`&`'来结束长的行是很方便的。

▶ **练习22.5**

怎样输入'**Fowl**'这行? (这太简单了。)

在本例中, 还有一个问题, 就是给出标题行, 它的格式与别的行是不一样的。在这种情况下, 格式的差别只在于字体是 slanted, 因此没什么特别难的; 我们只需要用

```
\sl American&\sl French&\sl Age&\sl Weight&\sl Cooking\cr
\sl Chicken&\sl Connection&\sl(months)&\sl(lbs.)&\sl Methods\cr
```

每次必须给出'`\sl`', 因为表格的每个单元都暗中封装在大括号中了。

在上面的表格中, 作者用'`\openup2pt`'来增加基线间的距离; 眼尖的读者还会发现在标题行和其它行之间也有额外的间距。这个额外的间距是由标题行紧后面的'`\noalign{\smallskip}`'插入的。一般地,

在 `\halign` 的任意 `\cr` 紧后面都可以使用

`\noalign{`⟨vertical mode material⟩`}`

TEX 将直接重复这些垂直模式的内容, 而不把它进行对齐, 并且当 `\halign` 结束时它就出现在此处。你可以象这里一样用 `\noalign` 插入额外间距, 或者插入控制分页的 penalty, 或者还可插入文本行(见第十九章)。在 `\noalign` 的大括号中的定义的影响局限在此组内。

命令 `\halign` 还可以用来调整栏之间的间距, 使得表格充满给定的区域。你不需要确定出栏间的间距是一个 quad; 可以让 TEX 来决定, 这是按照栏的宽度确定的, 因为 TEX 在栏之间放置了“制表粘连”。这个制表粘连一般是零, 但是你可以用‘`\tabskip=`⟨glue⟩’把它设置为任何所要的值。例如, 再讨论上面的表格, 但是把开头变成如下:

```
\tabskip=1em plus2em minus.5em
\halign to\hsize{\hfil\bf#&\hfil\it#\hfil&\hfil#\hfil&
          \hfil#\hfil&#\hfil\cr
```

表格的主体没有改动, 但是从导言中去掉了间距 `\quad`, 而用一个非零的 `\tabskip` 来代替。还有, ‘`\halign`’被改为‘`\halign to\hsize`’; 这意味着表格的每行都放在宽度为当前 `\hsize` 的值的盒子中, 即, 段落中通常的水平行宽度。所得的结果如下:

American Chicken	*French Connection*	*Age (months)*	*Weight (lbs.)*	*Cooking Methods*
Squab	*Poussin*	2	3/4 to 1	Broil, Grill, Roast
Broiler	*Poulet Nouveau*	2 to 3	1 1/2 to 2 1/2	Broil, Grill, Roast
Fryer	*Poulet Reine*	3 to 5	2 to 3	Fry, Sauté, Roast
Roaster	*Poularde*	5 1/2 to 9	Over 3	Roast, Poach, Fricassee
Fowl	*Poule de l'Année*	10 to 12	Over 3	Stew, Fricassee
Rooster	*Coq*	Over 12	Over 3	Soup stock, Forcemeat

一般地, TEX 把制表粘连放在第一栏之前, 最后一栏之后和对齐的各栏之间。用‘`\halign to`⟨dimen⟩’或‘`\halign spread`⟨dimen⟩’就可以得到最后的对齐尺寸, 就象使用‘`\hbox to`⟨dimen⟩’和‘`\hbox spread`⟨dimen⟩’一样。这个命令控制着制表粘连的设置; 但是它不影响栏单元中粘连的设置。(象以前讨论过的那样, 这些单元已经放在宽度为其自然宽度的盒子中了。)

因此, 如果制表粘连不能伸缩, 那么‘`\halign to \hsize`’得到的仅仅是松散或溢出的盒子而已。如果制表粘连不能收缩到给定尺寸, 就出现了溢出的盒子; 在这种情况下, 就会在终端和 log 文件中出现警告, 但是在输出结果中不会在超出尺寸的表格上标记黑方块。警告信息显示的是“模板行”(见第二十七章)。

上面给出的例子在所有地方用的都是同样的制表粘连, 但是可以在导言中重新设置 `\tabskip` 来改变它。 TEX 把在 `\halign` 后面的‘`{`’前读入的制表粘连放在第一栏前面, 把第一个模板后面‘`&`’前读入的

制表粘连放在第一和第二栏之间; 等等。在最后一个模板后面的 `\cr` 前读入的制表粘连放在最后一栏的后面。例如, 在

```
\tabskip=3pt
\halign{\hfil#\tabskip=4pt& #\hfil&
  \hbox to 10em{\hss\tabskip=5pt # \hss}\cr ...}
```

中, 导言中给出了对齐的行, 它包含下列七个部分:

制表粘连 3 pt;
第一栏, 模板为'`\hfil#`';
制表粘连 4 pt;
第二栏, 模板为'`#\hfil`';
制表粘连 4 pt;
第三栏, 模板为'`\hbox to 10em{\hss# \hss}`';
制表粘连 5 pt。

TEX 把模板复制下来而不解释它们, 但是要去掉任何 `\tabskip` 这个粘连。更确切地说, 导言的记号被直接送到模板而不进行宏展开; TEX 只寻找命令'`\cr`', '`&`', '`#`', '`\span`'和'`\tabskip`'。跟在'`\tabskip`'后面的 ⟨glue⟩ 按通常方法读入(进行宏展开), 并且相应的记号不包括在当前模板中。注意, 在上面的例子中, '`5pt`'后面的空格也被去掉了。`\tabskip=5pt` 出现在大括号中间, 但是它不能把此定义变成局域的, 因为 TEX 没有"看到"这些大括号; 类似地, 如果 `\tabskip` 前面有'`\global`', TEX 也不会生成全局定义, 它只把'`\global`'放在模板中。导言中所有 `\tabskip` 的赋值对 `\halign` 都是局域的(除非 `\globaldefs` 是正的), 因此当这个特殊的 `\halign` 结束时, `\tabskip` 的值又变成 3 pt 了。

当'`\span`'出现在导言中时, 就引起下一个记号被展开, 即在 TEX 读入前, "展-开-了"。

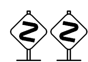
▶ **练习22.6**
为下列表格设计一个导言:

England	P. Philips	1560–1628	*Netherlands*	J. P. Sweelinck	1562–1621
	J. Bull	*c*1563–1628		P. Cornet	*c*1570–1633
Germany	H. L. Hassler	1562–1612	*Italy*	G. Frescobaldi	1583–1643
	M. Prætorius	1571–1621	*Spain*	F. Correa de Arauxo	*c*1576–1654
France	J. Titelouze	1563–1633	*Portugal*	M. R. Coelho	*c*1555–*c*1635

在每行的左右两边制表粘连为零; 在中间是 1 em plus 2 em; 在名字前面是 .5 em plus .5 em; 在日期前面是 0 em plus .5 em。假设每行是象这样输入的:

```
France&J. Titelouze&1563--1633&
  Portugal&M. R. Coelho&\\1555--\\1635\cr
```

其中已经把'`\\`'定义为'`\def\\{{\it c\/}}`'了。



▶ **练习22.7**

设计一个导言, 使得下列表格

rydw i = I am	**ydw i** = am I	**roeddwn i** = I was
rwyt ti = thou art	**wyt ti** = art thou	**roeddet ti** = thou wast
mae e = he is	**ydy e** = is he	**roedd e** = he was
mae hi = she is	**ydy hi** = is she	**roedd hi** = she was
rydyn ni = we are	**ydyn ni** = are we	**roedden ni** = we were
rydych chi = you are	**ydych chi** = are you	**roeddech chi** = you were
maen nhw = they are	**ydyn nhw** = are they	**roedden nhw** = they were

的输入方式象下面这样:

```
mae hi=she is&ydy hi=is she&roedd hi=she was\cr
```

▶ **练习22.8**
本表格右边的第二栏的断行是由 TEX 确定的, 这样第二栏正好是 16 em 宽。另外, 表格的行是象这样输入的:

```
\\393&Plato's {\sl Apology\/};
  Xenophon's
   {\sl Memorabilia\/};
  Aristophanes'
   {\sl Ecclesiazus\ae\/}\cr
```

你知道这个对齐中的导言是什么吗? [这些资料来自 Will Durant's *The Life of Greece* (Simon & Schuster, 1939).]

B.C.
397: War between Syracuse and Carthage
396: Aristippus of Cyrene and Antisthenes of Athens (philosophers)
395: Athens rebuilds the Long Walls
394: Battles of Coronea and Cnidus
*c*393: Plato's *Apology*; Xenophon's *Memorabilia*; Aristophanes' *Ecclesiazusæ*
391–87: Dionysius subjugates south Italy
391: Isocrates opens his school
390: Evagoras Hellenizes Cyprus
387: "King's Peace"; Plato visits Archytas of Taras (mathematician) and Dionysius I
386: Plato founds the Academy
383: Spartans occupy Cadmeia at Thebes
380: Isocrates' *Panegyricus*

有时候, 一个模板还需要对栏中的一到两个单元进行调整。例如, 在刚才给出的例子中, 第一栏的冒号由模板'`\hfil#:␣`'给出; 但是在此栏的第一个单元, 'B.C.', 却没有冒号。 TEX 允许从给定的模板跳出来, 方法如下: 如果对齐单元的第一个记号是'`\omit`'(在宏展开后), 那么就忽略掉导言的模板; 用空模板'`#`'来代替。例如, 上面表格中的'B.C.'是在导言后面直接输入'`\omit\hfil\sevenrm B.C.`'而得到的。可以在任何栏使用 `\omit`, 但是它必须是第一个; 否则 TEX 就插入导言中的模板。

如果你明了了 TEX 在处理 `\halign` 时要做的事情, 就会认识到使用某些命令的时机是十分重要的。当导言被读入时, 不进行宏展开, 除了前面讨论过的外; 但是一旦在导言结尾的 `\cr` 读入了, TEX 就必须向前看看是否后面跟的是 `\noalign` 或者是 `\omit`, 并且直到读入下一个非空格记号才进行宏展开。如果记号不是 `\noalign` 或 `\omit`, 它就回来再读入, 并且 TEX 开始读入模板(还要展开宏)。模板分两部分, 称为 u 和 v 部分, 其中 u 是在'`#`'的前面, v 是在它的后面。当 TEX 完成 u 部分时, 它就回到既不是 `\noalign` 也不是 `\omit` 的记号进行读入, 并且一直读到结束单元的 `&` 或 `\cr` 为止; 接着读入模板的 v 部分。一个叫 `\endtemplate` 的特殊内部命令总是放在 v 部分的结尾; 它使 TEX 把此单元放在一个"未设定的盒子"中, 此盒子的粘连在栏的最后宽度确定以后才设置。这样, TEX 就为读入下一个单元做好了准备; 它再次寻找 `\omit`(在 `\cr` 后面还要寻找 `\noalign`), 并且继续进行着同样的过程。

对于刚刚讨论的这个过程, 以 `\if...` 或展开为替换文本时第一个记号为 `\if...` 的任何宏为开头都有危险; 原因是, 在模板被读入后, 才进行条件测试。(当 `\if` 被展开时, TEX 还要寻找是否有 `\omit`。) 例如, 如果 `\strut` 被定义为

`\ifmmode`⟨text for math modes⟩`\else`⟨text for nonmath modes⟩`\fi`

并且如果 `\strut` 出现在对齐单元的开头, 那么即使在模板为'`$#$`'的时候, TEX 也把它展开为 ⟨text for nonmath modes⟩, 因为当 TEX 正在寻找可能出现的 `\omit` 时还未处在数学模式中。跟着可

能出现混乱。因此, 附录 B 中 `\strut` 的替换文本实际上是

```
\relax\ifmmode...
```

并且还把'`\relax`'放在所有会遇到这种时机问题的其它宏中。有时候在模板插入之前, 却希望 TEX 展开一个条件语句, 但是细心的宏编写者要注意什么情况下会出问题。

当你要排版的是数字表格时, 通常要把栏中的小数点对齐。例如, 如果如'0.2010'和'297.1'这两个数字同时出现在一个栏中, 那么你就希望得到的是'$\begin{matrix}0.2010\\297.1\end{matrix}$'。这个结果看起来不是特别舒服, 但是人们就是这样做的, 因此你可能也要遵守。得到这种结果的一种方法是把此栏看作两栏, 就象 `\eqalign` 把一个公式看作两个公式一样; 小数点'.'放在第二个半栏的开头。但是作者要用的是另一个更简单的方法, 它利用的是在大多数字体中, 数字 0, 1, . . ., 9 的宽度都是一样的: 你可以选定一个在表格中用不到的字符, 比如'`?`', 把它变成活动符, 用它生成与一个数字的宽度相当的空白。接下来, 不难把这样的空白字符放在表格单元中, 使得每栏可以是居中, 居左或居右的。例如, '`??0.2010`'和'`297.1???`'的宽度一样, 因此它们的小数点很容易对齐。下面过给出了为此设置'`?`'的一种方法:

```
\newdimen\digitwidth
\setbox0=\hbox{\rm0}
\digitwidth=\wd0
\catcode‘?=\active
\def?{\kern\digitwidth}
```

最后两个定义对某些组是局域的, 比如放在 `\vbox` 中, 这样当表格结束时'`?`'就恢复了其正常的性质。

我们来看看数学中的一些应用。首先假定你要把一个小表格:

$$\begin{matrix} n & = 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ \ 8\ \ 9\ \ 10\ 11\ 12\ 13\ \ 14\ \ 15\ \ 16\ \ 17\ \ 18\ \ 19\ \ \ 20\ \ \ldots \\ \mathcal{G}(n) & = 1\ 2\ 4\ 3\ 6\ 7\ 8\ 16\ 18\ 25\ 32\ 11\ 64\ 31\ 128\ 10\ 256\ \ 5\ \ 512\ 28\ 1024\ldots \end{matrix}$$

排版为列表方程。利用 `\eqalign` 或 `\atop` 这些笨办法太烦琐, 因为 $\mathcal{G}(n)$ 和 n 的数字个数并不总是相同。用导言 ⟨preamble⟩ 的方式可能跟漂亮:

```
$$\vbox{\halign{⟨preamble⟩\cr
     n\phantom)&0&1&2&3& ... &20&\dots\cr
     {\cal G}(n)&1&2&4&3& ... &1024&\dots\cr}}$$
```

另一方面, 导言 ⟨preamble⟩ 要足够长, 因为此表格有 23 栏; 因此好像 `\settabs` 和 `\+` 个简单。象这种导言通常有周期性结构的情况, TEX 有一个非常有帮助的特性, 并且如果把一个额外的'`&`'放在一个模板的紧前面, 那么 TEX 将把此导言看作一个无穷系列, 此系列从标记的模板开始一直到 `\cr` 为止。例如,

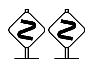
t_1 & t_2 & t_3 && t_4 & t_5 `\cr` 被看作 t_1 & t_2 & t_3 & t_4 & t_5 & t_4 & t_5 & t_4 & $\cdots$

和

&t_1 & t_2 & t_3 & t_4 & t_5 `\cr` 被看作 t_1 & t_2 & t_3 & t_4 & t_5 & t_1 & t_2 & t_3 & $\cdots$.

每个模板后面的制表粘连也随着此模板而重复。按照后面对齐单元实际用到的栏的数目, 导言延伸到所需要的长度。因此, 为了解决 $\mathcal{G}(n)$ 这个问题, 所有要设置的导言为

```
$\hfil#$ =&&\ \hfil#\hfil\cr
```



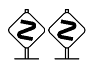
现在假设要排版是三对列表公式, 所有的 = 都要对齐:

$$\begin{aligned} V_i &= v_i - q_iv_j, & X_i &= x_i - q_ix_j, & U_i &= u_i, \qquad \text{for } i \ne j; \\ V_j &= v_j, & X_j &= x_j, & U_j &= u_j + \textstyle\sum_{i\ne j} q_iu_i. \end{aligned} \tag{23}$$

用三个 \eqalign 来做并不是那么简单的, 因为带下标'$i \ne j$'的 $\sum$ 把公式右边变得比别的公式要大; 基线不一致, 除非把"幻影"放在其它两个 \eqalign 公式中(见第十九章)。\eqalign 在附录 B 中是由 \halign 定义的宏, 我们不用 \eqalign, 而直接用 \halign。得到此列表公式的自然方法是使用

```
$$\vcenter{\openup1\jot \halign{⟨preamble⟩\cr
  ⟨first line⟩\cr ⟨second line⟩\cr}}\eqno(23)$$
```

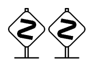
因为 \vcenter 把行放在一个盒子中, 这个盒子相对于方程编号'(23)'是正确居中的; 宏 \openup 在行之间插入了额外的间距, 基线第十九章中讨论的那样。



好了, 现在我们看看怎样输入第一行 ⟨first line⟩ 和第二行 ⟨second line⟩。通常约定把'&'放在我们要对齐的符号前面, 因此显然解决方法是使用

```
V_i&=v_i-q_iv_j,&X_i&=x_i-q_ix_j,&
  U_i&=u_i,\qquad\hbox{for $i\ne j$};\cr
V_j&=v_j,&X_j&=x_j,&
  U_j&=u_j+\sum_{i\ne j}q_iu_i.\cr
```

因此, 此对齐有六栏。我们可以把公共元素放在导言中(比如'V_'和'=v_'), 但是那样做更巧妙但是更容易出错。



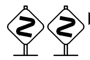
剩下的只是构造那些行的导言了。在等号 = 的左边, 我们要把左边的居右, 在等号 = 的右边, 要右边的居左。因为我们把公式分为两个部分了, 所以有点复杂, 但是我们要让结果中所出现的这两个部分看起来象一个公式。因为我们把'&'放在了关系符号紧前面, 所以在公式右边紧开头要插入'{}'; 这样 TeX 就可以在'${}=...$'的等号前面插入正确的间距了, 但是在 '$=...$'的等号前面没有插入间距。因此, 所要的导言为

```
$\hfil#$&${}#\hfil$&
  \qquad$\hfil#$&${}#\hfil$&
  \qquad$\hfil#$&${}#\hfil$
```

第三和第四栏与第一和第二栏基本一样, 只是多了一个分开方程的 \qquad; 第五和第六栏与第三和第四栏是一样的。我们可以再次用'&&'把此导言简化为

```
$\hfil#$&&${}#\hfil$&\qquad$\hfil#$
```

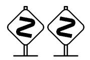
只要略微练习一下, 你就发现在需要导言时可以顺利地完成它。但是, 大多数文稿用不到它, 因此呆会你就会遇见这方面的一些练习。



▶ **练习22.9**
看看怎样得到下列列表公式:

$$10w + \ 3x + 3y + 18z = 1, \tag{9}$$

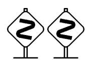
$$6w - 17x \qquad - \ 5z = 2. \tag{10}$$



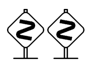
当表格的某些单元要占用两栏以上时, 情况就更复杂了。 TEX 提供了两种解决的方法。首先, 有一个命令 `\hidewidth`, plain TEX 把它定义为

```
\hskip-1000pt plus 1fill
```

换句话说, `\hidewidth` 有一个很负的“自然宽度”, 但是可伸长到无限远。如果在对齐中在某些单元右边放上 `\hidewidth`, 其效果就是, 忽略掉此单元的宽度, 并且让它可延伸出盒子右边。(回想一下; 当 `\halign` 确定栏宽度时, 此栏的宽度不是最宽的那个。) 类似地, 如果把 `\hidewidth` 放在单元的左边, 就伸出左边界; 并且你还可以把 `\hidewidth` 放在左右两边, 我们下面要讨论它。



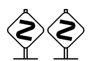
第二种方法是使用原始命令 `\span`, 它可以在表格的任意行中代替‘`&`’来使用。(我们已经知道, `\span` 在导言中表示“展开”; 但是在导言外面它的用法完全不同。) 当‘`\span`’出现在‘`&`’的位置上时, 在 `\span` 前后的内容都看作与普通文本一样, 但是要把它放在一个盒子中而不是两个盒子中。这个组合盒子的宽度是两个盒子各自的宽度加上它们之间的制表粘连; 因此, 合并的盒子与其它行未合并的盒子是对齐的。



例如, 假设有三栏, 模板分别是 u_1 `#` v_1 `&` u_2 `#` v_2 `&` u_3 `#` v_3; 假设栏宽度为 w_1, w_2, w_3; g_0, g_1, g_2, g_3 是设定粘连后的制表粘连的宽度; 出现在对齐中的行是

a_1`\span` a_2`\span` a_3`\cr`

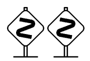
这样, ‘$u_1a_1v_1u_2a_2v_2u_3a_3v_3$’的内容(即, 栏 1 后面跟着栏 2 和栏 3 的内容)就被放在一个宽度为 $w_1 + g_1 + w_2 + g_2 + w_3$ 的 hbox 中。此盒子前面是宽度为 g_0 的粘连, 后面是宽度为 g_3 的粘连, 这个更大的盒子就是对齐的行。



可以把 `\omit` 与 `\span` 联合起来使用。例如, 如果继续使用上面的例子的记号, 那么行

`\omit` a_1 `\span` a_2 `\span\omit` a_3 `\cr`

就把内容‘$a_1u_2a_2v_2a_3$’放在刚刚讨论的 hbox 中。



因为合并几个栏并且忽略掉其模板是经常用到的, 所以 plain TEX 提供了一个宏 `\multispan`, 它把给定数目的栏合并起来。例如, ‘`\multispan3`’展开就是‘`\omit\span\omit\span\omit`’。如果要合并的栏的数目大于 9, 就要把它放在大括号中, 比如, ‘`\multispan{13}`’。

前一段太抽象了, 因此我们用一个例子看看 `\span` 实际上怎样用。假设你输入的是

```
$$\tabskip=3em
\vbox{\halign{&\hrulefill#\hrulefill\cr
    first&second&third\cr
    first-and-second\span\omit&\cr
    &second-and-third\span\omit\cr
    first-second-third\span\omit\span\omit\cr}}$$
```

此模板给出了任意多个模板'`\hrulefill#\hrulefill`'; 宏 `\hrulefill` 象 `\hfill` 一样, 只是把空白换为水平标尺了。因此, 所得到的对齐是被充满的, 它的输出为合并的栏:

first	second	third
first-and-second		____
____	_second-and-third_	
_____first-second-third_____		

当制表粘连把栏分开时, 标尺就断开了。在第一行看不到标尺, 因为在此行单元的宽度为栏的宽度。但是, 如果制表粘连为 1 em 而不是 3 em, 得到的表格是这样的:

first	_second_	third
first-and-second		____
____	second-and-third	
__first-second-third__		

▶ **练习22.10**

看看下列表格, 它叫做 Walter 工作表:

1	Adjusted gross income		$4,000
2	Zero bracket amount for		
	a single individual	$2,300	
3	Earned income	1,500 (underlined)	
4	Subtract line 3 from line 2		800 (underlined)
5	Add lines 1 and 4. Enter here		
	and on Form 1040, line 35		$4,800

定义一个导言, 使得下列输入就得到了 Walter 工作表。

```
\halign{⟨preamble⟩\cr
  1&Adjusted gross income\dotfill\span\omit\span&\$4,000\cr
  2&Zero bracket amount for&\cr
   &a single individual\dotfill\span\omit&\$2,300\cr
  3&Earned income\dotfill\span\omit&\underbar{ 1,500}\cr
```

```
4&Subtract line 3 from line 2\dotfill
    \span\omit\span&\underbar{ 800}\cr
5&Add lines 1 and 4. Enter here\span\omit\span\cr
 &and on Form 1040, line 35\dotfill\span\omit\span&\$4,800\cr}
```

(宏 `\dotfill` 与 `\hrulefill` 一样, 只是用圆点代替了标尺; 宏 `\underbar` 把其变量放在 hbox 中并在其下划线。)

注意, 在上一个练习中, 第二行中"过早"出现了 `\cr`。在对齐中, 每行的栏数不必相等; '`\cr`'表示当前行没有其它栏了。

▶ **练习22.11**
看看怎样输入下列通用矩阵: $\begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \multicolumn{4}{c}{\dotfill} \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$.

合并栏的出现使得 TeX 确定栏宽度的方法更复杂; 它不能直接把栏单元的宽度选定为最大自然宽度, 还必须考虑要确保能把足够大的合并栏放下。下面是 TeX 实际要做的事: 首先, 如果任意相邻的一对栏总是合并为一个单元(即, 只要用到一个, 它们之间就用了 `\span`。), 那么这两个栏就以一个栏而出现, 并且把它们之间的制表粘连设置为零。这其实是把问题简化为每个制表符的位置就是一个边界这种情况。我们假定简化后还有 n 个栏, 并且对 $1 \le i \le j \le n$, 设 w_{ij} 是从栏 i 到 j 合并后的所有单元的最大自然宽度; 如果没有这样的合并单元, 就设$w_{ij} = -\infty$。(栏的互相依存的出现保证了对每个 j, 存在 $i \le j$ 使得 $w_{ij} > -\infty$。) 设 t_k 为栏 k 和 $k+1$ 之间制表粘连的自然宽度, 其中 $1 \le k < n$。现在, 栏 j 最后的宽度由下列公式来确定:

$$w_j = \max_{1\le i\le j} \bigl(w_{ij} - \textstyle\sum_{i\le k<j} (w_k + t_k)\bigr)$$

其中依次 $j = 1, 2, \ldots, n$。因此, 如所要求的那样, 对所有 $i \le j$ 有 $w_{ij} \le w_i + t_i + \cdots + t_{j-1} + w_j$。在确定了宽度 w_j 后, 制表粘连可能要伸缩; 如果收缩, w_{ij} 可能就要比合并栏 i 到 j 的盒子的最后宽度要大, 从而在这个盒子中的粘连要收缩。

这些公式一般足以胜任, 但是有时候会得到意想不到的结果。例如, 假定 $n = 3$, $w_{11} = w_{22} = w_{33} = 10$, $w_{12} = w_{23} = -\infty$ 以及 $w_{13} = 100$; 也就是说, 栏本身很窄, 但是把这三个栏合并后的单元却很宽。在这种情况下, TeX 的公式为 $w_1 = w_2 = 10$, 但是 $w_3 = 80 - t_1 - t_2$, 这样所有剩下的宽度都留给了第三个栏。如果不希望是这样, 就要用 `\hidewidth`, 或者增加栏间制表粘连的自然宽度。

在制表中下一步出现的问题是绘制水平或垂直线。掌握了绘制表格线的人一般称得上是 TeX 大师了。你做好准备了吗?

如果用错误的方法来绘制垂直线, 就很麻烦; 但是恰好有一种方法可以巧妙地把它们绘制在表格中。第一步是用'`\offinterlineskip`', 它的意思是没有行间粘连; 在这种巧妙的方法中, TeX 不允许插入行间粘连, 因为每条线都假定包含一个 `\vrule`, 而它紧挨着上面和/或下面的另一条线的 `\vrule`。我们把一个支架(strut)放在每行中, 并把它放在导言中; 这样, 每条线就有了相应的高度和深度了, 并且不

需要用到行间粘连。 TEX 把对齐中的每个栏单元都放在一个 hbox 中, 而此盒子的高度和和深度被设置为整个行的高度和深度; 因此, 命令 `\vrule` 就伸长到此行的顶部和底部, 而不管它们的高度和/或深度是否给定。

一个“栏”要分配给每个垂直线, 并且这样的栏可以设定为模板‘`\vrule#`’。这样, 在对齐的正常行中, 通过把此单元直接变成空的就得到了垂直线; 如果要在某些行要忽略掉此线就用‘`\omit`’; 要得到非标准高度的线就用‘`height 10pt`’, 等等。

下面的小表格就举例说明了刚才的要点。[数据来自于 A. H. Westing, *BioScience* **31** (1981), 523–524。]

```
\vbox{\offinterlineskip
\hrule
\halign{&\vrule#&
  \strut\quad\hfil#\quad\cr
height2pt&\omit&&\omit&\cr
&Year\hfil&&World Population&\cr
height2pt&\omit&&\omit&\cr
\noalign{\hrule}
height2pt&\omit&&\omit&\cr
&8000\BC&&5,000,000&\cr
&50\AD&&200,000,000&\cr
&1650\AD&&500,000,000&\cr
&1850\AD&&1,000,000,000&\cr
&1945\AD&&2,300,000,000&\cr
&1980\AD&&4,400,000,000&\cr
height2pt&\omit&&\omit&\cr}
\hrule}
```

Year	World Population
8000 B.C.	5,000,000
50 A.D.	200,000,000
1650 A.D.	500,000,000
1850 A.D.	1,000,000,000
1945 A.D.	2,300,000,000
1980 A.D.	4,400,000,000

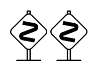
在本例中, 第一, 三, 五栏就是留给垂直线了。水平线由‘`\hrule`’生成, 方法是在 `\halign` 使用 `\hrule` 或者在对齐中用‘`\noalign{\hrule}`’, 这是因为 `\halign` 是在一个 vbox 中, 而此盒子的宽度是整个表格的宽度。水平线也可以在 `\halign` 用‘`\multispan5\hrulefill`’生成, 因为它得到的线横贯了五个栏。

此表格中唯一没搞清楚的就是那几个‘`height2pt&\omit&&\omit&\cr`’的行; 你知道它们的作用吗? 命令 `\omit` 的意思是, 此处没有数字的信息, 并且它还把支架 `\strut` 限制在行外面; ‘`height2pt`’得到了高为 2 pt 的第一个 `\vrule`, 并且其它两个垂直线将遵照此设定。因此, 就得到了伸长两个 pt 的垂直线, 从而它们就接到水平线上了。这一点点接上就使得盒子化的表格看起来很漂亮; 这也是高品质的表现。

▶ **练习22.12**

看看为什么此表格的每行都以‘`&\cr`’结束, 而不只是‘`\cr`’。

在表格中插入垂直线的另一种方法是先把表格排好版, 再退回来(利用负粘连)插入垂直线。



下面是另一个表格; 它已经成为一个经典例子, 从 Michael Lesk 把它作为其格式化表格的程序的报告中的第一个例子而出现时起 [Bell Laboratories Computing Science Technical Report **49** (1976)]。它举例说明了绘制线时出现的几个典型的问题。为了证明 TEX 能完成不同要求的表格, 下面用准备粘连来调整栏的宽度; 此表出现了两次, 一次是由'`\halign to125pt`'生成, 一次是由'`\halign to200pt`'生成, 其它的没有什么变化。

AT&T Common Stock		
Year	Price	Dividend
1971	41–54	$2.60
2	41–54	2.70
3	46–55	2.87
4	40–53	3.24
5	45–52	3.40
6	51–59	.95*

* (first quarter only)

AT&T Common Stock		
Year	Price	Dividend
1971	41–54	$2.60
2	41–54	2.70
3	46–55	2.87
4	40–53	3.24
5	45–52	3.40
6	51–59	.95*

* (first quarter only)

此表用下列方法得到:

```
\vbox{\tabskip=0pt \offinterlineskip
\def\tablerule{\noalign{\hrule}}
\halign to⟨dimen⟩{\strut#& \vrule#\tabskip=1em plus2em&
  \hfil#& \vrule#& \hfil#\hfil& \vrule#&
  \hfil#& \vrule#\tabskip=0pt\cr\tablerule
&&\multispan5\hfil AT\&T Common Stock\hfil&\cr\tablerule
&&\omit\hidewidth Year\hidewidth&&
 \omit\hidewidth Price\hidewidth&&
 \omit\hidewidth Dividend\hidewidth&\cr\tablerule
&&1971&&41--54&&\$2.60&\cr\tablerule
&&   2&&41--54&&2.70&\cr\tablerule
&&   3&&46--55&&2.87&\cr\tablerule
&&   4&&40--53&&3.24&\cr\tablerule
&&   5&&45--52&&3.40&\cr\tablerule
&&   6&&51--59&&.95\rlap*&\cr\tablerule \noalign{\smallskip}
&\multispan7* (first quarter only)\hfil\cr}}
```

要讨论的要点是: (1). 第一栏包含一个支架(strut); 否则就要在'AT&T'和'(first quarter only)'这些行中放支架, 因为这些行把所有其它栏可能内置的支架的模板给忽略掉了。(2). 在标题栏使用了'`\hidewidth`', 这样栏的宽度只受数字的宽度的影响。(3). 使用'`\rlap`'是为了不让星号影响数字对齐。(4). 如果把制表粘连从'`1em plus2em`'变成'`0em plus3em`', 对齐也不会伸出右边, 因为'AT&T Common Stock'比所有合并栏的自然宽度都大; 剩下的宽度都放在'Dividend'的栏中。

▶ **练习22.13**

看看怎样在‘AT&T Common Stock’的上下增加 2 pt 的间距。

▶ **练习22.14**

排版下列图表, 它的宽度正好是 36em:

J. H. Böhning, 1838

M. J. H. Böhning, 1882

M. D. Blase, 1840

L. M. Bohning, 1912

E. F. Ehlert, 1845

P. A. M. Ehlert, 1884

C. L. Wischmeyer, 1850

如果在调试对齐时有问题, 有时候可以把‘`\ddt`’放在导言的模板开头和结尾, 这样会有帮助。它是一个未定义的控制系列, 使 TEX 停下来, 把模板剩下的内容显示出来。当 TEX 停止时, 可以用 `\showlists` 或其它命令来看看计算机是怎样处理的。如果 TEX 不停下来, 就是说它从来就没用到模板的这部分。

可以在对齐中使用对齐。因此, 当 TEX 遇见‘`&`’, ‘`\span`’或者‘`\cr`’时, 需要确定是哪一个对齐。规则是, 当‘`&`’, ‘`\span`’或‘`\cr`’出现在与当前单元开始同样级别的大括号时, 此单元就结束了; 即, 在每个单元中必须有相同数目的左右大括号。例如, 在行

```
\matrix{1&1\cr 0&1\cr}&\matrix{0&1\cr 0&0\cr}\cr
```

中, 当 TEX 扫描到 `\matrix` 的变量时, 不会回到第一栏的模板, 因为在此变量中的 `&` 和 `\cr` 都被封装在大括号中了。类似地, 如果所得到的模板没有相同数目的左右大括号, 那么导言中的 `&` 和 `\cr` 也不表示模板的结束。

使用 `&`, `\span` 和 `\cr` 时要小心, 因为这些记号会被 TEX 扫描截取, 即使它不展开宏。例如, 如果在对齐单元的中间使用‘`\let\x=\span`’, 那么 TEX 就认为‘`\span`’是单元的结束, 这样 `\x` 就变得与模板中跟在‘`#`’后面的第一个记号相同了。可以把它放在大括号中来把它隐藏起来, 比如‘`{\global\let\x=\span}`’。(附录 D 中讨论了怎样避免在这里使用 `\global`。)

有时候会把对齐的最后一行的 `\cr` 落掉了。这会出现不可理解的效果, 因为 TEX 不是千里眼。例如, 看看下面这个直白的例子:

```
\halign{\centerline{#}\cr
  A centered line.\cr
  And another?}
```

(注意, 落掉了 `\cr`。) 当态处理这个错误的行时会出现古怪的情况, 因此要注意。模板以‘`\centerline{`’开头, 因此 TEX 开始扫描 `\centerline` 的变量。因为在问号后面没有‘`\cr`’, 所以问号后面的‘`}`’被看作 `\centerline` 的变量结束, 而不是 `\halign` 的结束。 TEX 不会认为此对齐结束了, 除非随后的文本中

有‘...{...\cr’这样的格式。的确, 象‘a}b{c’这样的单元对模板‘\centerline{#}’是合理的, 因为它得到的是‘\centerline{a}b{c}’; 因此在这种情况下 TEX 不给出错误信息是对的。但是在当前情况下, 计算机的思维和用户的想法是不同的, 因此几行以后就会出现令人不知所措的错误信息。

为了避免这样的情况, 有一个原始命令 \crcr, 它象 \cr 一样, 只是当紧跟在 \cr 或 \noalign{...} 后面时就不起作用了。这样, 当编写象 \matrix 这样的宏时, 就可以在用户的变量结尾处安全地插入 \crcr; 如果用户落掉最后面的 \cr, 就可补救此错误, 如果用户没落掉, 也没有什么影响。

是不是已经对键入 \cr 感到厌烦了? 可以用下列方法让 plain TEX 在每输入行的结尾处自动插入 \cr:

```
\begingroup \let\par=\cr \obeylines %
\halign{⟨preamble⟩
  ⟨first line of alignment⟩
  ...
  ⟨last line of alignment⟩
  }\endgroup
```

这是因为 \obeylines 把 ASCII 的 ⟨return⟩ 变成了活动符, 把它当前作为 \par 来使用, 并且 plain TEX 在每个输入行的结尾有一个 ⟨return⟩(见第八章)。如果不想要某个行结尾的 \cr, 只需要输入‘%’, 相应的 \cr 就被“注释”掉了。(这种特殊的方法不能处理行 \+, 因为 \+ 是一个宏, 其变量由记号‘\cr’来作为分界符, 不能直接用用同样意思的记号来代替 \cr。但是如果需要, 也可以通过重新定义 \+ 来绕过这个障碍。例如, 定义宏 \alternateplus, 它与 \+ 一样, 只是它的变量把活动符 ^^M 当做分界符; 这样, 把命令‘\let\+=\alternateplus’包括为 \obeylines 一部分即可。)

控制系列 \valign 与 \halign 类似, 但是行和栏的规则要改变。在这种情况下, \cr 是栏的底部, 并且对齐的栏是 vbox, 它们在水平模式下放在一起。每栏的各个单元是深度为零的 vbox(即, 就象 \boxmaxdepth 是零, 在第十二章讨论过); \valign 的每行的单元高度是同一格式中最大的高度, 就象 \halign 的每栏单元的宽度是最大宽度一样。命令 \noalign 现在用来在栏之间插入水平模式的内容; 命令 \span 合并的是行。一般人们在使用 TEX 至少一年以后才发现要用到 \valign; 并且一般只是单行格式‘\valign{\vfil#\vfil\cr...}’。但是如果要用到, 基本原理已经有了。

23. 输出例行程序

在第十五章, 我们讨论了 TEX 组建页面的方法, 它分为两个基本阶段: TEX 收集超过一个页面的内容; 接着按照页面之间的最佳断点分出一个页面; 再接下来用同样的方法收集下一个页面的内容。页码, 页眉等类似的内容在分页后由一个 TEX 的一系列特殊命令添加上去, 这就是所谓的当前**输出例行程序**。

Plain TEX 有一个输出例行程序, 由它完成普通的任务。它处理大多数文稿所要求的简单任务, 以及象 \footnote 和 \topinsert 所做的插入对象这些更复杂的任务, 就象第十五章中"危险"段落中讨论的那样。在本章我们将首先讨论怎样直接改变 plain TEX 输出例行程序的设置; 接着将转到怎样定义输出例行程序的详细内容, 以处理更复杂的任务。

如果你不加修改地运行 plain TEX 的格式, 得到的页码就在底部; 并且每个页面的宽为 $8\frac{1}{2}$ 英寸, 高为 11 英寸, 在四个边缘是 1 英寸的页边。这种格式对专业论文的预印比较合适, 但是你可能希望改变它, 特别是在你要得到的不是专业论文的预印格式时。

例如, 在第六章的实战中我们已经看到, 页面中内容的宽度由水平行尺寸 \hsize 的不同值来确定。在 plain TEX 的格式中, 由设置'\hsize=6.5in'得到的 8.5 英寸页面中, 页边为 1 英寸; 你可以按自己的要求改变 \hsize。同样, 要改变页面的垂直尺寸就要改变 \vsize。Plain TEX 设置 \vsize=8.9in(不是 9in, 因为 \vsize 没有把每个页面底部页码的空间包括进去); 如果给出'\vsize=4in', 就得到更短的页码, 每页只用了 4 英寸。除了在任务最开头或者已经把所有页面从 TEX 的内存中去掉后以外, 最好别乱改动 \hsize 和 \vsize。

如果要改变你的输出结果最后打印的位置, 可以把它偏移, 即给 \hoffset 和 \voffset 一个非零值。例如,

```
\hoffset=.5in   \voffset=1.5in
```

就把输出从正常位置向右平移半英寸, 向下平移 1.5 英寸。注意别偏移得超出所用打印的物理介质的边缘, 除非你知道这样超出边界不会出现问题。

TEX 常常用于排版公告, 小册子或者其它没有页码的文档。如果在文稿开头声明了

```
\nopagenumbers
```

那么 plain TEX 就不在每个页码的底部插入数字了。

实际上, \nopagenumbers 是控制页眉和页脚的更广泛的方法的一个特殊情形。Plain TEX 的输出例行程序把一个叫*页眉 headline* 的特殊的行放在每个页面的顶部, 把另一个叫*页脚 footline* 的特殊行放在底部。此页眉一般是空的, 而页脚一般是居中的页码, 但是通过重新定义控制系列 \headline 和 \footline, 可以得到你所要的页眉和页脚。例如,

```
\headline={\hrulefill}
```

将把水平线放在每个页眉的顶部。基本思路是, plain TEX 把'\line{\the\headline}'放在顶部, 把'\line{\the\footline}'放在底部, 在这些额外行和其它内容之间是一个空行。(记住, \line 的定义

是‘`\hbox to\hsize`’; 因此, 页眉和页脚被放在宽度为正常页面行宽的盒子中。) `\headline` 的正常值是‘`\hfil`’, 这样就没有可见的页眉。前面给出的宏 `\nopagenumbers` 的定义就是‘`\footline={\hfil}`’。

`\footline` 的正常值是‘`\hss\tenrm\folio\hss`’; 它把页码居中, 字体为 `\tenrm`, 因为 `\folio` 是得到以文本格式表示的当前页码的控制系列。

就象第十五章中讨论的那样, 页码在 TEX 的内部寄存器 `\count0` 中, 并且 plain TEX 把 `\pageno` 定义为 `\count0`。因此, 如果要让输出的下一页的页码为 100, 就要给出 ‘`\pageno=100`’。宏 `\folio` 把负的页码变成 Roman 数字; 如果你的文稿以‘`\pageno=-1`’开始, 那么页码就是 i, ii, iii, iv, v 等等。实际上, 附录 B 把 `\folio` 定义为

```
\ifnum\pageno<0 \romannumeral-\pageno \else\number\pageno \fi
```

重要的是把所定义页码和页脚的每个字体名称明确包括进来, 因为 TEX 中的输出例行程序在某些预料不到的时候要用到。例如, 假定 `\footline` 已经设定为‘`\hss\folio\hss`’, 而没有使用 `\tenrm`; 这样, 当 TEX 要输出页码时, 页码就用任何当前字体进行排版。在这样的情况下, 就会出现预想不到的结果, 因为当输出页面 100 时 TEX 最可能在页面 101 的中间。

▶ **练习23.1**
看看怎样把短破折号放在页码两边。例如, ‘ – 4 – ’出现在页面 4 的底部。

下面是页眉的一个例子, 其中页码放在顶部。还有, 奇数和偶数页面要做不同的对待:

```
\nopagenumbers % suppress footlines
\headline={\ifodd\pageno\rightheadline \else\leftheadline\fi}
\def\rightheadline{\tenrm\hfil RIGHT RUNNING HEAD\hfil\folio}
\def\leftheadline{\tenrm\folio\hfil LEFT RUNNING HEAD\hfil}
\voffset=2\baselineskip
```

习惯上, 英文书籍的奇数页面在右边, 偶数页面在左边。作为某些页上的页眉的文本常常称为“可变页眉”。当使用页眉时, 象本例这样用 `\voffset` 向下偏移两行文字的空间一般是比较明智的, 这样在所得到页面的顶部仍然是一英寸的页边。

▶ **练习23.2**
假定要用 TEX 排版个人简历 résumé, 它包含几个页面。看看怎样定义 `\headline` 才能使第一页的页眉为居中的黑体‘**RÉSUMÉ**’, 而其它页眉是这样:

Résumé of A. U. Thor . . . . Page 2

如果不修改 `\vsize`, 那么所有的页眉和页脚都出现在同样的位置, 而不管它们中间页面的内容如何。因此, 例如, 如果象第十五章中讨论的那样使用了 `\raggedbottom`, 从而导致每个页面所包含的内容长度不尽相同, 那么页脚上面就变得参差不齐; 页脚也不会向上移动。如果改变了 `\vsize`, 那么页脚的位置就会相应地改变, 而页面不变。

本章剩下的内容把对象限制在要得到与 plain TEX 所给出的输出例行程序不同的格式的人。随后的所有段落都有双“危险”标记, 因为在攻入本语言最后的堡垒时要精通 TEX 的其它内容。第二十二章教出了 TEX 大师, 即, 能用 `\halign` 和 `\valign` 排版出复杂表格的人; 接下来的内容将使你一鼓作气地达到统帅的等级, 即, 可以设计输出例行程序的人。当你学会后, 就会欣喜地发现, 就象对齐方式一样, 输出例行程序不象刚开始看起来那样神秘。

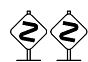
我们首先来回顾一下第十五章结尾的一些规则。 TEX 周期性地输出一个页面的内容, 这是通过把主垂直列在所选定的最佳处裂分而得到的, 并且在这时它进入内部垂直模式并且开始读入当前 `\output` 例行程序的命令。当输出例行程序开始时, `\box255` 中包含 TEX 已经完成的页面; 输出例行程序可以看作对此 vbox 进行一些处理。当输出例行程序结束时, 在内部垂直模式下构建的项目列就放在分页后面的内容的紧前面。用这种方法, TEX 的分页结果会被明显修改: 已完成分页的页面的某些或全部内容会被去掉并且送到下一页。

当前 `\output` 例行程序被定义为一个记号列参数, 就象 `\everypar` 或者 `\errhelp` 一样, 只是 TEX 自动在开头插入组开始符号, 在结尾插入组结束符号而已。这些编组符号有利于输出例行程序不受分页时 TEX 的状态的干扰; 例如, 当输出例行程序在页面上放置页眉和页脚时, 常常修改 `\baselineskip`, 这样额外的大括号就使得这个修改被限制在局部中。如果没有给出 `\output` 例行程序, 或者用户声明了‘`\output={}`’, 那么 TEX 就使用自己的例行程序, 它实际上等价于‘`\output={\shipout\box255}`’; 它输出的页面没有页眉和页脚, 也没有页码。

实际上是 TEX 的原始命令 `\shipout`⟨box⟩ 产生的输出。它把盒子的内容送到 TEX 的主要输出文件——`dvi` 文件; 在 TEX 运行结束后, `dvi` 文件就包含了一个紧凑的与设备无关的指令编码, 它准确地给出了要输出的信息。当盒子被送出时, TEX 把 `\count0` 到 `\count9` 的值都显示在你的终端上, 就象第十五章中讨论的那样; 这十个计数器都保存在 `dvi` 文件中, 它们就确定了此页面。当盒子被送出时, 在 ⟨box⟩ 中出现的所有 `\openout`, `\closeout` 和 `\write` 命令都按照它们的正常顺序被执行。因为命令 `\write` 展开一个宏, 就象第二十一章讨论的那样, 所以当 `\shipout` 正在进行时 TEX 的扫描程序可能会发现语法错误。如果在 `\shipout` 时 `\tracingoutput` 不为零, 那么要被送出的 ⟨box⟩ 的内容将用符号的形式写入你的 log 文件。可以在任何地方使用 `\shipout`, 而不仅仅是输出例行程序中。

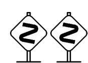
`\write` 延迟性强加了一个值得注意的限制: 当命令 `\shipout` 给定时, 必须确保出现在 `\write` 文本中的所有宏要有正确的定义。例如, 附录 B 中的 plain TEX 格式暂时把空格变成活动符, 并且声明‘`\global\let␣=\space`’; 这是因为 `\obeyspaces` 可能在 `\write` 命令中起作用, 因此把 ␣ 定义为活动符应该持续到下一个 `\shipout`, 即使此时 TEX 可能不再用空格作活动符。

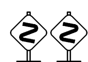
第十五章提到, 在输出例行程序开始之前, TEX 把特殊值赋予给了包括 `\box255` 在内的某些内部寄存器和参数。插入对象被放在它自己的 vbox 中, 并且 `\insertpenalties` 被设为与延迟的插入对象的总数相等; 还有, 参数 `\outputpenalty` 被设为当前断点的 penalty 的值。当这些量赋予特殊值后, 输出例行程序就可以用来做很多特殊的工作。例如, plain TEX 的输出例行程序用到一个命令 `\supereject`(它把所有延迟的插入对象都驱出), 原因是 `\supereject` 把 `\outputpenalty` 变成 -20000, 并且用 `\insertpenalties` 来确定是否所有的插入对象都被延迟了。

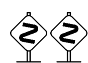

默认的输出例行程序‘\shipout\box255’给出的是一个极端的情况, 它不把任何内容放在下一个页面的垂直列中。另一个极端是

```
\output={\unvbox255 \penalty\outputpenalty}
```

它什么也不输出, 并且把所有内容都放回主垂直列。(命令‘\unvbox255’把这个页面从盒子中取出, 命令‘\penalty\outputpenalty’在选定的断点处重新插入 penalty。) 这使得整个页面与其后的内容恰好合并起来, 因为当 TEX 调用 \output 例行程序时没有丢掉任何断点处的粘连和 penalty; 这样 TEX 就回来重新考虑分页。如果 \vsize 没有改变, 并且所有的插入对象都还在原地, 那么就得到同样的分页; 但是找到它的速度比以前要更快, 因为垂直列已经构建好了——不需要再次分段为行了。当然, 这样的输出例行程序使得 TEX 陷入死循环, 因此它仅仅是个极端的例子, 而没有什么用处。


为了防止这样的循环, 在出现这种情况时要对你的输出例行程序进行一些改进。如果你出现了失误, TEX 可能会帮你找出错误, 因为内置了一个特殊的循环检测方法: 有一个叫 \deadcycles 的内部整数变量, 在每次 \shipout 后都回到零, 并且在每个 \output 紧前面要增加 1。这样, \deadcycles 跟踪的就是最近一次 \shipout 后输出例行程序启动的次数, 只要你自己不改变 \deadcycles 的值。还有一个整数参数叫做 \maxdeadcycles, plain TEX 把它设置为 25。当输出例行程序将要启动时(即当 \deadcycles 要增加时), 如果 \deadcycles 大于或等于 \maxdeadcycles, TEX 就输出错误信息, 并且执行默认的输出例行程序, 来代替你的例行程序。


当输出例行程序结束时, \box255 应该被置空。换句话说, 你必须用一些方法来处理此盒子中的信息; 它应该被送出或者是放在某些其它地方。类似地, 当 TEX 准备放入下一页内容, 在启动输出例行程序紧前面, \box255 应该被置空。如果在某个这些时刻 \box255 没有被置空, 那么 TEX 将告诉说你误用了这个特殊寄存器, 并且此寄存器的内容将被毁掉。


但是我们现在别再讨论边界上的情况和特殊参数了; 我们来看看一些真实的例子。在附录 B 中的 plain TEX 的输出例行程序被设置为‘\output={\plainoutput}’, 其中, \plainoutput 被定义为

```
\shipout\vbox{\makeheadline
  \pagebody
  \makefootline}
\advancepageno
\ifnum\outputpenalty>-20000 \else\dosupereject\fi
```

下面我们逐行讨论这个“程序”:

1) 宏 \makeheadline 构建了一个高度和深度都为零的 vbox, 这样页眉就正好放在页面的空白处了。它的实际定义是:

```
\vbox to 0pt{\vskip-22.5pt
  \line{\vbox to8.5pt{}\the\headline}\vss}
\nointerlineskip
```

−22.5 pt 这个神秘的常数等于 (顶部粘连 − 支架高度 − 2(行高)), 即 10 pt − 8.5 pt − 24 pt; 它把页眉的基点正好放在页面顶行基点上面正好 24 pt 的地方, 只要页眉或顶行不是太宽即可。

2) 宏 `\pagebody` 的定义是

```
\vbox to\vsize{\boxmaxdepth=\maxdepth \pagecontents}
```

`\boxmaxdepth` 的值设置为 `\maxdepth`, 这样就构建了 vbox, 其中假定 TEX 的页面已经设置为 `\box255` 了。

3) 宏 `\pagecontents` 得到了一个垂直列, 它的所有内容都是页面的主体, 即 `\box255` 的内容以及图表(插入在顶部)和脚注(插入在底部):

```
\ifvoid\topins \else\unvbox\topins\fi
\dimen0=\dp255 \unvbox255
\ifvoid\footins\else % footnote info is present
  \vskip\skip\footins
  \footnoterule
  \unvbox\footins\fi
\ifraggedbottom \kern-\dimen0 \vfil \fi
```

这里的 `\topins` 和 `\footins` 是 plain TEX 中用到的两种插入对象的类代码; 如果要添加更多插入对象的类, `\pagecontents` 应该随之改变。注意, 这些盒子要去掉外围盒子, 这样来自插入对象的粘连才能变成主页面的粘连。附录 B 中的宏 `\footnoterule` 把一个分隔线放在页面和脚注之间; 它的垂直列净高度是 0 pt。设置底部不对齐要插入无限大粘连, 它的级别要比 `\topskip` 的伸长能力高。

4) 宏 `\makefootline` 把 `\footline` 放在相应的位置:

```
\baselineskip=24pt
\line{\the\footline}
```

5) 宏 `\advancepageno` 一般要把 `\pageno` 加 +1; 但是如果 `\pageno` 是负的(Roman 数字), 就加 −1。新的 `\pageno` 值在下一次输出例行程序中被调用。

```
\ifnum\pageno<0 \global\advance\pageno by-1
\else \global\advance\pageno by 1 \fi
```

6) 最后, 宏 `\dosupereject` 是为了把所有延迟的插入对象清除掉, 不管它们是图表还是脚注:

```
\ifnum\insertpenalties>0
  \line{} \kern-\topskip \nobreak
  \vfill\supereject\fi
```

这里的负 `\kern` 是为了抵消空 `\line` 上面出现的粘连 `\topskip` 的自然间距; 此空行的盒子是防止 `\vfill` 在分页时丢失。`\dosupereject` 所得到的垂直列被放在 TEX 的列中以便下次使用, 就在延迟的插入对象象在第十五章中讨论的那样被重新考虑后。因此, 出现了另一个 `\supereject`, 并且这个过程一直持续到没有插入对象为止。

▶ **练习23.3**

看看怎样改变 plain TEX 的输出例行程序才能得到两倍的页面。原来放在页面 1, 2, 3 等上的内容现在放在页面 1, 3, 5 等上面; 偶数页面除了页眉和页脚之外是空白。(就把它看作以后要把照片贴在这些空白页上。)

假定现在要得到双栏的格式。更准确地说, 我们要修改 plain TEX 使得它的每栏宽度为 \hsize=3.2in。所输出的每个页面应该包含两个这样的栏, 中间有 0.1 in 的间距; 这样, 每页的文本区仍然是 6.5 英寸宽。页眉和页脚要横贯两个栏, 但是栏自己要象书籍对开的页面那样包含各自的插入对象。换句话说, 每个栏有自己的脚注和图表; 我们不必改变宏 \pagebody。

为了完成这个任务, 我们首先引入新的尺寸寄存器 \fullhsize, 它表示整个页面的宽度。

```
\newdimen\fullhsize
\fullhsize=6.5in \hsize=3.2in
\def\fullline{\hbox to\fullhsize}
```

宏 \makeheadline 和 \makefootline 应该变成用‘\fullline’而不是‘\line’。

新输出例行程序将用控制系列 \lr 来设置‘L’或‘R’, 根据下一栏在下一页面的左边还是右边而定。当左栏结束时, 输出例行程序直接把它保存在盒子寄存器中; 当右栏结束时, 例行程序输出两个栏并且增加页码。

```
\let\lr=L \newbox\leftcolumn
\output={\if L\lr
    \global\setbox\leftcolumn=\columnbox \global\let\lr=R
  \else \doubleformat \global\let\lr=L\fi
  \ifnum\outputpenalty>-20000 \else\dosupereject\fi}
\def\doubleformat{\shipout\vbox{\makeheadline
    \fullline{\box\leftcolumn\hfil\columnbox}
    \makefootline}
  \advancepageno}
\def\columnbox{\leftline{\pagebody}}
```

宏 \columnbox 使用 \leftline 是为了确保它得到的盒子的宽度是 \hsize。但是在输出例行程序的开头, \box255 的宽度通常——但不总是——等于 \hsize; 任何其它宽度都会把格式毁坏。

当双栏设置完毕时, 最后一栏处在左边有一半的几率, 因此仍然不能输出。在这种情况下, 代码

```
\supereject
\if R\lr \null\vfill\eject\fi
```

补上一个空白的右栏, 以确保所有的内容都被输出。在最后一页, 可以把两栏对齐, 但是如果还要容纳下脚注和其它插入对象就要有些技巧。附录 E 中有用来对齐附录 I 中索引结尾的宏, 并且在页面中间开始一个双栏格式。

▶ **练习23.4**
怎样修改上面的例子才能得到三栏的输出?

因为 TEX 的输出例行程序滞后于其构建页面的动作, 所以如果要无限制地改动 `\headline` 或者 `\footline` 就会得到错误的结果。例如, 假定要排版一本书, 你所使用的格式允许在页面中间开始一章; 这样, 如果在开始新的章时改动可变的页眉是不对的, 因为实际的页面可能还没有包括新章的任何内容。再看看排版字典或会员册; 设计精良的参考书在每页或每对开页的顶部都给出了当前内容的区间, 使得读者在查找单个单词或人名时很容易翻阅。但是 TEX 非同步的输出方法使得难以找出当前页上内容的区间, 即使可以找见。

因此, TEX 给出了一种方法, 把"标记"放在列中; 这些标记告诉输出例行程序每个页面内容的区间。一般思路是, 可以在你所排版的内容中间声明

`\mark{`⟨mark text⟩`}`

其中 ⟨mark text⟩ 是一个记号列, 展开后是命令 `\edef`, `\message` 等等。 TEX 把这些标记文本的内部表示放在正在建立的列中; 接着随后, 当整个页被封装在 `\box255` 中时, TEX 允许输出例行程序调用本页的第一和最后一个标记文本。

理解它的最好方法可能是把它想像为 TEX 输出的一个任意长的垂直列, 其中有盒子, 粘连和象 penalty 和标记这样的其它项目。由于某种原因, 整个长的垂直列被分割为页面, 并且这些页面一次只有一个进入输出例行程序。只要页面被放在 `\box255` 中, TEX 设定了本质上与宏一样的三个量的值:

- `\botmark` 是在刚刚放在盒子中的页面上最近出现的标记文本;
- `\firstmark` 是在刚刚放在盒子中的页面上第一次出现的标记文本;
- `\topmark` 的值为当前页面盒子紧前面的 `\botmark` 的值。

在第一个页面前, 所有这三个值都是空的, 即它们展开时什么也没有。当在页面上没有标记时, 所有三个标记都等于前一个 `\botmark`。

例如, 假定你的文稿包括四个标记, 并且分页方式如下: `\mark{`α`}` 出现在第 2 页, `\mark{`β`}` 和 `\mark{`γ`}` 在第 4 页, `\mark{`δ`}` 在第 5 页。这样,

页面	`\topmark` 为	`\firstmark` 为	`\botmark` 为
1	null	null	null
2	null	α	α
3	α	α	α
4	α	β	γ
5	γ	δ	δ
6	δ	δ	δ

当在垂直模式下使用 `\mark` 命令时, TEX 把一个标记放在主垂直列中。当在水平模式下使用 `\mark` 命令时, TEX 把它看作象 `\vadjust` 和 `\insert` 那样垂直模式下的内容; 即, 在分段为行后,

每个标记在主垂直列中的位置就在它最初出现的行的紧后面。如果在受限水平模式下使用 \mark, 那么标记就用与 \insert 和 \vadjust 同样的方法移出封装的垂直列(见第二十四章); 但是在盒子中埋得太深的标记不会移出, 因此它不会作为 \firstmark 或 \botmark 而出现。类似地, 出现在内部垂直模式下的 \mark 将放在一个 vbox 中, 并且它也不进入主垂直列。

第十五章讨论了命令 \vsplit, 它本身就可以分割垂直列。这个命令有时候可以用在 TeX 普通的页面构建上。例如, 如果要直接把某些内容放在等高的两个栏中, 就可以把它们放在一个 vbox 中, 接着用 \vsplit 把盒子分为两半; 根本不需要任何输出例行程序。命令 \vsplit 设置了两个类似于宏的量, 它们在第十五章没有被讨论: \splitfirstmark 和 \splitbotmark 就延伸为最近的 \vsplit 命令所裂分的垂直列中出现的第一和最后一个标记文本。如果没有这样的标记, 它们就是空的。\topmark, \firstmark, \botmark, \splitfirstmark 和 \splitbotmark 的值是全局的; 即, 它们不受 TeX 编组的影响。

大多数字典在对开页面的顶部用 \firstmark 和 \botmark 配对来给出引导词。例如, 如果单词‘type’的定义起于第 1387 页, 并且延续到第 1388 页, 那么在 1387 页(页在右边)的引导词为‘type’; 但是在 1388 页(页在左边)的引导词为字典中的下一个词(比如, ‘typecast’), 即使 1388 页上还有‘type’的内容。

字典的程序可以很好地处理字典, 因为读者每次是从头开始查阅字典的。但是对象作者的专业书籍 *Art of Computer Programming* 需要用不同的方法, 其中第 1.2.8 节(例如)从第 78 页的中间开始, 但是第 78 页的顶部包含有第 1.2.7 节的练习 19–24。第 78 页的页眉指的是‘1.2.7’, 这样对查找练习 1.2.7–22 有好处。注意, 字典的约定是把 1.2.8 放在第 78 页, 但是只有当第 1.2.8 节从此页顶部就开始这才比较合适。

继续讨论这个来自 *The Art of Computer Programming* 的例子, 假定第 1.2.8 节的文稿以一个宏开始, 象下面这样:

```
\beginsection 1.2.8. Fibonacci Numbers.
```

那么怎样定义 \beginsection? 下面是一种尝试:

```
\def\beginsection #1. #2.
  {\sectionbreak
  \leftline{\sectionfont #1. #2}
  \mark{#1}
  \nobreak\smallskip\noindent}
```

宏 \sectionbreak 提示 TeX 在此处分页或者留下一定的间距; 比如, \sectionbreak 可以定义为‘\penalty -200 \vskip18pt plus4pt minus6pt’。宏 \beginsection 结尾的命令把本节第一段的缩进给去掉了。但是我们关心的是跟在 \leftline 后面与输出例行程序有关的命令 \mark。在本例中我们要讨论的是, 在盒子把此节的题目包含进去的紧后面, 我们要在第 1.2.8 节的开头把‘\mark{1.2.8}’插入到主垂直列中。

这样一个 \mark 就行了吗? 很不幸, 不行, 即使我们为了简化而假设每个页面上至多有一节也不行。包含第 1.2.8 节开头的页面就有 \topmark=1.2.7 和 \firstmark=1.2.8, 而不管节是不是从

页面顶部开始。在这个应用中我们要的是 \topmark 和 \firstmark 之间的一个交叉: 反映的应该是此页面第一行紧后面情况的标记文本。 而 TEX 没有给出它。

解决方法是把 \mark 放在 \sectionbreak 紧前面而不是 \leftline 紧后面。这样 \topmark 反映的总是当前页面顶行的节的情况。(想一想。)

但是, *The Art of Computer Programming* 的格式比这更复杂。在左侧页面上, 页眉上的节名反映的是页眉顶部的情况, 就象我们刚讨论的那样, 但是在右侧页面上, 要对应的是此页面底部的情况。前一个问题的解决之道是正确地把 \topmark 放在顶部, 而它却错误地把 \botmark 放在底部。为了满足这两个要求, 必须把更多的内容赋予此标记。下面就是一种解决的方法:

```
\def\beginsection #1. #2.
  {\mark{\currentsection \noexpand\else #1}
  \sectionbreak
  \leftline{\sectionfont #1. #2}
  \mark{#1\noexpand\else #1} \def\currentsection{#1}
  \nobreak\smallskip\noindent}
\def\currentsection{} % the current section number
```

思路就是引入两个标记, 一个放在节断点紧前面, 一个放在节开始的紧后面。还有, 每个标记有两个部分; 在第 1.2.7 节和第 1.2.8 节的潜在断点紧前面的标记是'1.2.7\else 1.2.8', 而潜在断点后面的是'1.2.8\else 1.2.8'。这样, 对应于页面底部的节号是 \botmark 的左分量; 对应于页面顶部的节号是 \topmark 的右分量。宏 \rightheadline 可以利用'\iftrue\botmark\fi'来读入左分量, 宏 \leftheadline 可以用'\expandafter\iffalse\topmark\fi'来读入右分量。

▸ **练习23.5**
B. C. Dull 使用的定义非常类似与上面的那个, 但是他把第二个 \mark 从 \leftline 紧后面变成紧前面。这样会出现上面错误?

▸ **练习23.6**
前面构造的标记的格式为'α\else$\beta$', 其中 α 和 β 是两个不相关的内容。用'\else'使得可以用 \iftrue 和 \iffalse 来选择 α 或 β。推广一下: 假定在一个任务中, 标记中有五个不相关的内容, 并且每个标记的格式为'α_0\or$\alpha_1$\or$\alpha_2$\or$\alpha_3$\or$\alpha_4$'。看看怎样从这样的标记中任意选择出某个 α。

在输出例行程序最后, 我们讨论把它应用于索引, 比如本手册中出现在附录 I 中的索引。在这样的索引中, 最复杂的条目是象这样的:

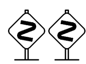
Main entry, 4, 6, 8–10, 12, 14–16,
  18–22, 24–28, 30.
 first subsidiary entry, 1–3, 6, 10–11,
  15, 21, 24, 28.
 second subsidiary entry, 1, 3, 6–7,

10, 15, 21, 25, 28, 31.

主条目和子条目都是作对齐, 第一行后要悬挂缩进两个 em; 子条目的第一行要缩进一个 em。我们的目的是从下列这样的输入来排版:

```
\beginindex
...
Main entry, 4, 6, 8--10, 12, 14--16, 18--22, 24--28, 30.
\sub first subsidiary entry, 1--3, 6, 10--11, 15, 21, 24, 28.
\sub second subsidiary entry, 1, 3, 6--7, 10, 15, 21, 25, %
  28, 31.
...
\endindex
```

其中'...'表示其它条目。每个输入行一般给出一个主条目或一个子条目; 如果条目太长在一行放不下, 就可以在此行结尾输入'␣%'使得可以在下一行继续输入。

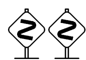

对这个索引问题, 感兴趣的是设置一个标记系统, 使得当一个条目出现在不同栏或页面时输出例行程序可以插入一行特殊文字。例如, 如果在上述六行中出现了分页, 那么输出例行程序可以输出特殊文字行

Main entry (*continued*):

并且当分页出现在子条目时, 也会出现另外的特殊行

subsidiary entry (*continued*):

下面的解决方法得到的标记使得当断点出现在主条目之间时 `\botmark` 为空; 如果出现在六个例句行的第 1, 2 或 4 行后面时它是'`Main entry`'; 当出现在第 3 行(第一个条目中)后面时它是'`Main entry\sub first subsidiary entry`'; 如果出现在第 5 行后面时它是'`Main entry\sub second subsidiary entry`'。


读者可能希望在看答案之前解决这个问题, 因为它没有什么难点, 比较简单。(继续: 先定义一个宏 \beginindex, 它左对齐, 并且生成给定的标记。在回到前一页, 在看答案前仔细研究这个问题。)

```
\def\beginindex{\begingroup
  \parindent=1em \maxdepth=\maxdimen
  \def\par{\endgraf \futurelet\next\inxentry}
  \obeylines \everypar={\hangindent 2\parindent}
  \exhyphenpenalty=10000 \raggedright}
\def\inxentry{\ifx\next\sub \let\next=\subentry
  \else\ifx\next\endindex \let\next=\vfill
  \else\let\next=\mainentry \fi\fi \next}
\def\endindex{\mark{}\break\endgroup}
\let\sub=\indent \newtoks\maintoks \newtoks\subtoks
\def\mainentry#1,{\mark{}\noindent
  \maintoks={#1}\mark{\the\maintoks}#1,}
\def\subentry\sub#1,{\mark{\the\maintoks}\indent
  \subtoks={#1}\mark{\the\maintoks\sub\the\subtoks}#1,}
```

即使你看完这个答案, 也希望有一些解释, 因为它用到了本手册前面未出现过的“TEX 技术”。

1) 宏 \beginindex 用 \begingroup 来保持其它改变的局部性; 因此, 当索引结束时, 不需要恢复 \parindent 和 \maxdepth 等以前的值。参数 \maxdepth 设定为 \maxdimen, 它实际上是无限大, 这样 \box255 的深度就是它所包含的最后盒子的真实深度; 下面我们将用到这个结果。(用这种方法可以安全地把 \maxdepth 无用化, 因为索引中的条目可以假定有相当小的深度。) 注意, 这里使用了 \obeylines, 因此在输入行的结尾处都插入了 \par。改变 \par 意思可以让它完成比以前更多的任务: 首先, 它实现了 \endgraf 这个 TEX 的普通 \par 的任务; 其次, 它把 \next 设置为下一行的第一个记号, 在此后面, 将展开宏 \inxentry。

2) 当 \inxentry 开始时,它要看 \next 以决定要做什么。有三种情况: 如果 \next 是‘\sub’, 此行就看作子条目; 如果 \next 是‘\endindex’, 下一个执行的命令就是‘\vfill\mark{}\break\endgroup’; 否则, 此行就是主条目。

3) 主条目的文本被放在 \mainentry 的参数 #1 中; 此参数的分界符是逗号。\mainentry 做的第一件事是‘\mark{}’, 断点在条目之间时就清空此标记。接着是‘\noindent’, 它使 TEX 进入水平模式并且放置粘连 \parskip。(粘连 \parskip 是行间的一个合理断点; 当排版好主条目的第一行时, 接下来的是行间粘连。) 接着另一个 \mark 自己被放在段落中; 它包含主条目的文本和一个叫做 \maintoks 的 \toks 寄存器, 此寄存器用来禁止标记文本展开。当段落结束和裂分为行时, 这个特殊的标记就立即放到段落第一行的后面, 因此如果分页出现在段落中的任何地方, 它就是 \botmark。

4) 类似的构造也用在 \subentry 上, 但是这个标记更复杂。\maintoks 寄存器包含的仍然是主条目。子条目的文本用另一个记号列寄存器 \subtoks 来添加。因为 \sub 已经定义得等于 \indent, 所以在此标记中它不会展开。

刚才定义的宏就把包含必要的标记的条目排版出来了; 现在我们必须构造一个输出例行程序, 它按照所要求的方法使用这些标记, 以插入象前面提到的新行‘(*continued*)’。我们再次建议读者在看答案之前动手解决这个问题。

```
\output={\dimen0=\dp255 \normaloutput
  \expandafter\inxcheck\botmark\sub\end}
\def\inxcheck#1\sub#2\end{\def\next{#1}%
  \ifx\next\empty % do nothing if \botmark is null
  \else\noindent #1\continued % ‘Main entry (continued):’
    \def\next{#2}%
    \ifx\next\empty % nothing more if \botmark has no \sub
    \else\let\sub=\continued \indent #2\fi
    \advance\dimen0 by-\prevdepth \kern\dimen0 \fi}
\def\continued{ ({\it continued}\thinspace):\endgraf}
```

这段代码比一般的要深奥。它假定了 \normaloutput 被送到 \box255(可能它是一个多栏的格式), 并且增加页码; 接着出现新的素材, 它由 \inxcheck 来执行。宏 \inxcheck 用一种有趣的方式来调用, 它允许把 \botmark 分解为其分量。如果 \botmark 为空, 那么 \inxcheck 的变量 #1 就是空的; 因此 \next 就等价于 \empty。(Plain TEX 声明‘\def\empty{}’是为了这种情况而使用。) 如果 \botmark 不包含记号 \sub, 那么变量 #1 就是 \botmark 的内容, 而 #2 为空。否则, 如果 \botmark 的形式为 α\sub$\beta$, 那么变量 #1 就是 α 而 #2 就是‘β\sub’。

如果 \botmark 是非空的, 那么宏 \inxcheck 就得到一行或多行文本, 它们将被放置到 TEX 主垂直列的页面断点的地方。并且最深奥的要点是在下面: 在分页处将有一个行间粘连, 它是基于断点前面的盒子深度而计算出的。对输出例行程序, 此深度是已知的, 因为它是 \box255 的深度。(\maxdepth 的值就是为此而变成无限大的。) 因此, 宏 \inxcheck 可以插入一个 \kern 以弥补旧盒子和已经计算好要插入到行间粘连前的盒子之间深度的差。没有此 \kern 的话, 间距就是错误的。在设计在随机输出行之间插入新盒子的输出例行程序前, 读者应该好好研究一下这个例子, 以理解命令 \kern 背后的原因。

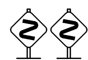
▶ **练习23.7**
修改这个构造, 使得在双栏时, 续行只插入在偶数页码的左侧栏中。

▶ **练习23.8**
真假题: 在本例中的宏 \inxcheck 至多把两行输出放在主垂直列。

当 TEX 遇到命令 \end 时, 只有当主垂直列已经完全输出了并且 \deadcycles=0时它才终止任务。 否则它就把与

```
\line{} \vfill \penalty-'10000000000
```

等价的命令插入到主垂直列中, 并且准备着再次读入记号‘\end’。其结果是重复调用输出例行程序直到所有内容都送出去了。特别是, 双栏格式中的最后一栏也不会落掉。



可能设计出一个输出例行程序, 它总是在主垂直列上剩下一些残余, 然而却不允许 `\deadcycles` 增加。在这种情况下, TEX 就永不结束! 输出例行程序可意识到, 它由 TEX 的终结程序所调用, 因为特殊的 `\penalty-'10000000000` 会导致 `\outputpenalty` 是很大的负值。这时, 如果必要, 输出例行程序就修改其参数使得出现一个不错的结尾。

24. 垂直模式汇总

前面的章节讨论了所有的 TEX 语言; 我们已经最后结束了这个以前未知领域的学习。万岁! 胜利了! 现在该系统地总结一下我们所学习的知识: 用条理化的方式来讨论这些结果, 而不是用一些非正式的例子的应用——就象我们已经学习的那样。小孩子在知道正式的语法规则前就已经会说话了, 但是当长大后, 语法规则就能派上用场了。本章——以及后面两章——的目的是简明扼要地总结 TEX 的语法, 这样熟练的用户就能尽可能高效地掌握它了。

在这些章中我们只涉及 TEX 的*原始*命令, 而不是大多数人所使用的 plain TEX 格式的高级命令。因此, 初学者应该跳过第 24–26 章, 感到需要时再进行学习。附录 B 是 plain TEX 的汇总, 以及希望知晓 TEX 用法的大多数人需要的一个参考指南。从高级方面了解 TEX 的最好方法是转到附录 B 的开始。

但是, 我们下面的目的是概括 TEX 的低级命令——更高级的宏结构就是建立在它上面的, 这是为的确需要知道这些详细知识的人提供一个参考。本章剩下的内容用小字排版, 意思与前面的一样, 因为它的内容与其它章标记双“危险”标志的差不多。我们不再重复地使用“危险”标识, 而直接把第 24–26 章都定义为“危险”的。

实际上 TEX 有几个特性看起来不值得在前面的章节提及, 因此这里为了完整而介绍一下它们。如果前面讨论的与下面要讲述的有些出入, 那么本章以及后面的章节就更正确一些。

我们将讨论 TEX 的消化过程, 即 TEX 是怎样处理已经到达它的“胃”中的记号列。第七章讲述了输入文件怎样在 TEX 的“嘴”中变成记号列, 还有第二十章讨论了 TEX 的可展开的记号怎样通过类似于反刍的过程在 TEX 的“食道”中转变为不可展开的记号。当不可展开的记号最好到达胃肠后, 就开始真正进行排版了, 我们在这些汇总章节中总结的就是它们。

每个到达 TEX 胃中的记号都被看作计算机要遵守或执行的一个命令。例如, 字母‘`L`’是用当前字体排版一个‘L’的命令; ‘`\par`’告诉 TEX 此段落结束。 TEX 总是处在第十三章讨论的六种模式之一, 并且在不同模式下一个命令的意思有时候也不同。本章讨论的是垂直模式(和内部垂直模式——它们几乎是一样的): 我们将讲述的是, 当每个原始命令出现在垂直模式中时, TEX 会做出什么反应。第二十五和二十六章用类似的方法讨论了水平模式和数学模式, 但是那些章比本章要短, 因为很多命令在所有模式下的意思是一样的; 对这些命令毋需重复三次, 一次即可。

有些命令有变量。换句话说, 命令后面的一个或多个记号可能会改变命令的意思, 并且那些记号并不被看作命令本身。例如, 当 TEX 执行对应于‘`\dimen2=2.5pt`’的记号序列时, 它只把第一个记号‘`\dimen`’看作命令; 下一个记号作为运算部分而执行, 因为 TEX 要知道 `\dimen` 寄存器所设定的 ⟨dimen⟩ 值是多少。

我们将用 1960 年 John Backus 和 Peter Naur 为计算机语言定义而引入的语法符号的变形来定义 TEX 的语言。在角括号中的量被看作文字, 或者它们由*语法规则*来定义出来, 这些语法规则给出了这个量由其它量构成的方法。例如,

⟨unit of measure⟩ ⟶ ⟨optional spaces⟩⟨internal unit⟩
　　| ⟨optional `true`⟩⟨physical unit⟩

把 ⟨unit of measure⟩ 定义为 ⟨optional spaces⟩ 后面跟一个 ⟨internal unit⟩, 或者定义为 ⟨optional `true`⟩ 后面跟 ⟨physical unit⟩。语法规则中的符号‘ ⟶ ’的意思是“的定义为”, ‘ | ’的意思是“或者”。

有时候语法规则是递归的, 此时右边的定义中包括了要被定义的量。例如,

⟨optional spaces⟩ ⟶ ⟨empty⟩ | ⟨space token⟩⟨optional spaces⟩

这个规则把与语法有关的量 ⟨optional spaces⟩ 定义为 ⟨empty⟩, 或者定义为一个 ⟨space token⟩ 后面跟 ⟨optional spaces⟩。量 ⟨empty⟩ 表示“什么也没有”, 即, 根本没有任何记号; 因此, 刚刚给出的语法规则的意思是, ⟨optional spaces⟩ 表示的是零个或多个空格。

另外, 在语法规则右边的角括号中, 不必全是量。也可以用显示记号。例如,

⟨plus or minus⟩ ⟶ $+_{12}$ | $-_{12}$

这个规则的意思是, ⟨plus or minus⟩ 表示类代码为 12 的字符记号, 加号或减号。

对关键词使用特殊的约定, 因为关键词的实际语法有点专用。字体为 typewriter 的字母, 象‘`pt`’这样, 意思是

⟨optional spaces⟩⟨p or P⟩⟨t or T⟩,

其中, ⟨p or P⟩ 表示 `p` 或者 `P` 的任一非活动符记号(与类代码无关), ⟨t or T⟩ 类似。

象‘`\dimen`’这样的控制系列用在下面的语法规则中时, 它表示当前意思与运行 TeX 时 `\dimen` 相同的任何记号。用 `\let` 或 `\futurelet` 也可赋予其它记号与这一样的意思, 并且控制系列 `\dimen` 自己的意思也可以被用户重新定义, 但是语法规则不去注意这些; 它只把‘`\dimen`’用作表示 TeX 的特殊原始命令的一种方法。(这个表示与‘$\boxed{\texttt{dimen}}$’不同, 后者表示实际名称为 `dimen` 的控制系列的记号; 见第七章。)

控制系列有时候乔装成字符, 只要它们的意思是由 `\let` 或 `\futurelet` 指定的。例如, 附录 B 声明了

```
\let\bgroup={   \let\egroup=}
```

并且这些命令把 `\bgroup` 和 `\egroup` 变得象左右大括号的作用一样。这样的控制系列称为“隐字符”; 当 TeX 把它们当做命令使用时, 按照与字符同样的方法来解释, 但是当作为命令的变量出现时却不总是这样。例如, 在上面给出的语法规则 ⟨plus or minus⟩ 中, 命令‘`\let\plus=+`’不会用 `\plus` 完全代替字符记号‘$+_{12}$’; 命令 ‘`\let\p=p`’ 也不会用 `\p` 完全来代替关键词 `pt` 的相应部分的。当 TeX 语法中显式和隐式字符都可以时, 下面的规则明确而仔细地给出了判断。

在上面 ⟨optional spaces⟩ 的语法中使用的量 ⟨space token⟩ 表示一个显式或隐式空格。换句话说, 它或者表示类代码为 10 的一个字符记号, 或者表示当前意思由 `\let` 或 `\futurelet` 设置为等于这样的记号的一个控制系列或活动符。

为了方便, 用符号‘{’, ‘}’和‘$’来表示类代码分别为 1, 2 和 3 的任何显式和隐式字符记号, 而不管实际字符代码是大括号或美元符号。这样, 例如, plain TeX 的 `\bgroup` 是一个‘{’的例子, 并且因此记号是‘$\{_1$’和‘$(_1$’; 但不是‘$\{_{12}$’。

这最后几段可以总结为, TeX 的正式语法规则右边的另一种方法由下列一项或多项组成: (1). 象 ⟨optional spaces⟩ 这样的符合语法的量; (2). 象 $+_{12}$ 这样的显式字符记号; (3). 象 `pt` 这样的关键词; (4). 象 `\dimen` 这样的控制系列名称; 或者, (5). 象 {, }, $ 这样的特殊符号。

在讨论 TeX 的语法的开头, 我们首先讨论作为命令的变量而经常出现的量——象 ⟨number⟩, ⟨dimen⟩ 和 ⟨glue⟩——的确切含义。最重要的是 ⟨number⟩, 它给出了一个整数值。下面是 ⟨number⟩ 的确切意思:

⟨number⟩ ⟶ ⟨optional signs⟩⟨unsigned number⟩
⟨optional signs⟩ ⟶ ⟨optional spaces⟩
  | ⟨optional signs⟩⟨plus or minus⟩⟨optional spaces⟩
⟨unsigned number⟩ ⟶ ⟨normal integer⟩ | ⟨coerced integer⟩
⟨normal integer⟩ ⟶ ⟨internal integer⟩
  | ⟨integer constant⟩⟨one optional space⟩
  | '$_{12}$⟨octal constant⟩⟨one optional space⟩
  | "$_{12}$⟨hexadecimal constant⟩⟨one optional space⟩
  | `$_{12}$⟨character token⟩⟨one optional space⟩
⟨integer constant⟩ ⟶ ⟨digit⟩ | ⟨digit⟩⟨integer constant⟩
⟨octal constant⟩ ⟶ ⟨octal digit⟩ | ⟨octal digit⟩⟨octal constant⟩
⟨hexadecimal constant⟩ ⟶ ⟨hex digit⟩ | ⟨hex digit⟩⟨hexadecimal constant⟩
⟨octal digit⟩ ⟶ 0_{12} | 1_{12} | 2_{12} | 3_{12} | 4_{12} | 5_{12} | 6_{12} | 7_{12}
⟨digit⟩ ⟶ ⟨octal digit⟩ | 8_{12} | 9_{12}
⟨hex digit⟩ ⟶ ⟨digit⟩ | A_{11} | B_{11} | C_{11} | D_{11} | E_{11} | F_{11}
  | A_{12} | B_{12} | C_{12} | D_{12} | E_{12} | F_{12}
⟨one optional space⟩ ⟶ ⟨space token⟩ | ⟨empty⟩
⟨coerced integer⟩ ⟶ ⟨internal dimen⟩ | ⟨internal glue⟩

⟨number⟩ 的值是相应的 ⟨unsigned number⟩ 的值, 对 ⟨optional sign⟩ 中每个负号乘以 −1。字母常数表示在 ⟨character token⟩ 中的字符代码; TeX 不会展开这个记号, 此记号应该是一个(字符代码, 类代码)对, 或者是活动符, 或者是由单个字符组成的控制系列。(至于 TeX 不展开的所有记号, 第二十章中有所有情形的完整列表。)⟨integer constant⟩ 后面不能紧接着一个 ⟨digit⟩; 换句话说, 如果几个数字连续着出现, 就把它们都看作同一个 ⟨integer constant⟩ 的各个部分。对 ⟨octal constant⟩ 和 ⟨hexadecimal constant⟩ 这样的量也是类似的。只有当 ⟨one optional space⟩ 必须存在时, 它从是 ⟨empty⟩, 即, 如果那里没有一个 ⟨space token⟩, TeX 为寻找 ⟨one optional space⟩ 要读入一个记号并且导致阻塞。

▶ **练习24.1**

你知道为什么你可能把'A_{12}'看作一个 ⟨hex digit⟩, 而字母 A 的类代码为 11 吗? (如果你不知道, 也没关系。)

现在, ⟨number⟩ 的定义除了三个量 ⟨internal integer⟩, ⟨internal dimen⟩ 和 ⟨internal glue⟩ 外就已经都讨论过了, 这三个量在后面讨论; 它们表示的是象参数和寄存器这样的对象。例如, `\count1`, `\tolerance` 和 `\hyphenchar\tenrm` 是内部整数; `\dimen10`, `\hsize` 和 `\fontdimen6\tenrm` 是内部尺寸; `\skip100`, `\baselineskip` 和 `\lastskip` 是内部粘连值。假定单位是大像素点后, 内部尺寸就变成整数了。例如, 如果 `\hsize=100pt` 并且 `\hsize` 以 ⟨number⟩ 出现, 那么它表示整数值 6553600。类似地, 如果把内部粘连值限制为一个尺寸(忽略伸缩性), 那么它也可以变成整数, 从而表示此尺寸。

现在我们转而讨论 ⟨dimen⟩ 及其伴生的 ⟨mudimen⟩ 的语法:

⟨dimen⟩ ⟶ ⟨optional signs⟩⟨unsigned dimen⟩
⟨unsigned dimen⟩ ⟶ ⟨normal dimen⟩ | ⟨coerced dimen⟩
⟨coerced dimen⟩ ⟶ ⟨internal glue⟩
⟨normal dimen⟩ ⟶ ⟨internal dimen⟩ | ⟨factor⟩⟨unit of measure⟩
⟨factor⟩ ⟶ ⟨normal integer⟩ | ⟨decimal constant⟩
⟨decimal constant⟩ ⟶ $\texttt{.}_{12}$ | $\texttt{,}_{12}$
  | ⟨digit⟩⟨decimal constant⟩
  | ⟨decimal constant⟩⟨digit⟩
⟨unit of measure⟩ ⟶ ⟨optional spaces⟩⟨internal unit⟩
  | ⟨optional `true`⟩⟨physical unit⟩⟨one optional space⟩
⟨internal unit⟩ ⟶ `em` ⟨one optional space⟩ | `ex` ⟨one optional space⟩
  | ⟨internal integer⟩ | ⟨internal dimen⟩ | ⟨internal glue⟩
⟨optional `true`⟩ ⟶ `true` | ⟨empty⟩
⟨physical unit⟩ ⟶ `pt` | `pc` | `in` | `bp` | `cm` | `mm` | `dd` | `cc` | `sp`

⟨mudimen⟩ ⟶ ⟨optional signs⟩⟨unsigned mudimen⟩
⟨unsigned mudimen⟩ ⟶ ⟨normal mudimen⟩ | ⟨coerced mudimen⟩
⟨coerced mudimen⟩ ⟶ ⟨internal muglue⟩
⟨normal mudimen⟩ ⟶ ⟨factor⟩⟨mu unit⟩
⟨mu unit⟩ ⟶ ⟨optional spaces⟩⟨internal muglue⟩ | `mu` ⟨one optional space⟩

当'`true`'出现时, 因子要乘以 1000 并且除以参数 `\mag`。物理单位在第十章中定义; `mu` 在第十八章中讨论过。

在掌握了量 ⟨number⟩, ⟨dimen⟩ 和 ⟨mudimen⟩ 的准确语法的基础上, 我们来讨论 ⟨glue⟩ 和 ⟨muglue⟩:

⟨glue⟩ ⟶ ⟨optional signs⟩⟨internal glue⟩
  | ⟨dimen⟩⟨stretch⟩⟨shrink⟩
⟨stretch⟩ ⟶ `plus` ⟨dimen⟩ | `plus` ⟨fil dimen⟩ | ⟨optional spaces⟩
⟨shrink⟩ ⟶ `minus` ⟨dimen⟩ | `minus` ⟨fil dimen⟩ | ⟨optional spaces⟩
⟨fil dimen⟩ ⟶ ⟨optional signs⟩⟨factor⟩⟨fil unit⟩⟨optional spaces⟩
⟨fil unit⟩ ⟶ `fil` | ⟨fil unit⟩ `l`
⟨muglue⟩ ⟶ ⟨optional signs⟩⟨internal muglue⟩
  | ⟨mudimen⟩⟨mustretch⟩⟨mushrink⟩
⟨mustretch⟩ ⟶ `plus` ⟨mudimen⟩ | `plus` ⟨fil dimen⟩ | ⟨optional spaces⟩
⟨mushrink⟩ ⟶ `minus` ⟨mudimen⟩ | `minus` ⟨fil dimen⟩ | ⟨optional spaces⟩

TeX 有大量可设置的内部量, 这样格式设计者就可以控制 TeX 的性质了。下面是所有这些量, 只是没有参数(它在后面列出来)。

⟨internal integer⟩ ⟶ ⟨integer parameter⟩ | ⟨special integer⟩ | `\lastpenalty`
  | ⟨countdef token⟩ | `\count`⟨8-bit number⟩ | ⟨codename⟩⟨8-bit number⟩
  | ⟨chardef token⟩ | ⟨mathchardef token⟩ | `\parshape` | `\inputlineno`
  | `\hyphenchar`⟨font⟩ | `\skewchar`⟨font⟩ | `\badness`
⟨special integer⟩ ⟶ `\spacefactor` | `\prevgraf`
  | `\deadcycles` | `\insertpenalties`
⟨codename⟩ ⟶ `\catcode` | `\mathcode`
  | `\lccode` | `\uccode` | `\sfcode` | `\delcode`
⟨font⟩ ⟶ ⟨fontdef token⟩ | `\font` | ⟨family member⟩
⟨family member⟩ ⟶ ⟨font range⟩⟨4-bit number⟩
⟨font range⟩ ⟶ `\textfont` | `\scriptfont` | `\scriptscriptfont`
⟨internal dimen⟩ ⟶ ⟨dimen parameter⟩ | ⟨special dimen⟩ | `\lastkern`
  | ⟨dimendef token⟩ | `\dimen`⟨8-bit number⟩
  | ⟨box dimension⟩⟨8-bit number⟩ | `\fontdimen`⟨number⟩⟨font⟩
⟨special dimen⟩ ⟶ `\prevdepth` | `\pagegoal` | `\pagetotal`
  | `\pagestretch` | `\pagefilstretch` | `\pagefillstretch`
  | `\pagefilllstretch` | `\pageshrink` | `\pagedepth`
⟨box dimension⟩ ⟶ `\ht` | `\wd` | `\dp`
⟨internal glue⟩ ⟶ ⟨glue parameter⟩ | `\lastskip`
  | ⟨skipdef token⟩ | `\skip`⟨8-bit number⟩
⟨internal muglue⟩ ⟶ ⟨muglue parameter⟩ | `\lastskip`
  | ⟨muskipdef token⟩ | `\muskip`⟨8-bit number⟩

⟨countdef token⟩ 是一个控制系列记号, 其中控制系列的当前意思由 `\countdef` 来定义; 其它的 ⟨dimendef token⟩ 等量也是类似定义的。⟨fontdef token⟩ 指的是由 `\font` 来定义, 或者它是叫做 `\nullfont` 的预定义字体标识符。当 ⟨countdef token⟩ 作为内部整数使用时, 它表示的是相应 `\count` 寄存器的值, 并且对 ⟨dimendef token⟩, ⟨skipdef token⟩ 和 ⟨muskipdef token⟩ 也是类似的。当 ⟨chardef token⟩ 和 ⟨mathchardef token⟩ 作为整数使用时, 它表示在 `\chardef` 和 `\mathchardef` 本身中的值。⟨8-bit number⟩ 是取值在 0 和 $2^8-1=255$ 之间的 ⟨number⟩; ⟨4-bit number⟩ 也是类似的。

只有在水平模式下, TeX 从允许 `\spacefactor` 是内部整数; 只有在垂直模式下, `\prevdepth` 可以取为内部尺寸; 只有当前数学列以数学粘连下面结尾时的数学模式下, `\lastskip` 可以是 ⟨internal muglue⟩; 在这样的情况下, `\lastskip` 不能是 ⟨internal glue⟩。当 `\parskip` 作为内部整数出现时, 它只表示所控制的行数, 而不是它们的尺寸和缩进。当当前页面不包含盒子时, 有七个特殊尺寸 `\pagetotal`, `\pagestretch` 等等都是零, 并且此时 `\pagegoal` 为 `\maxdimen` (见第十五章)。

由刚才给出的语法规则, 是可以准确知道在哪些数字量之间要有空格: TeX 允许 ⟨number⟩ 或

⟨dimen⟩ 前面有任意多个空格, 而后面至多跟一个空格; 但是, 在以不可展开的控制系列为结尾的 ⟨number⟩ 或 ⟨dimen⟩ 后面没有可选的空格。例如, 如果 TEX 在寻找一个 ⟨number⟩ 时遇见了'`\space\space24\space\space`, 那么它会扔掉前三个空格而保留下第四个; 类似地, 当'`24pt\space\space`', '`\dimen24\space\space`'和'`\pagegoal\space`'被看作 ⟨dimen⟩ 的值时, 只保留一个空格。

▶ **练习24.2**
'`24\space\space pt`'是一个合理的 ⟨dimen⟩ 吗?

▶ **练习24.3**
当 TEX 把'`+\baselineskip`', '`- -\baselineskip`'和'`1\baselineskip`'看作 ⟨glue⟩ 时, 它们之间有区别吗?

▶ **练习24.4**
假定在 plain TEX 中, 当 `\spacefactor` 等于 1000 时, `"DD DDPLUS2,5 \spacefactor\space` 得到的 ⟨glue⟩ 是什么?

现在我们转而讨论 TEX 的参数, 在前一章的某个地方已经见过一些; 把它们都集中在一起讨论很方便。⟨integer parameter⟩ 是下列某个记号:

`\pretolerance` (badness tolerance before hyphenation)
`\tolerance` (badness tolerance after hyphenation)
`\hbadness` (badness above which bad hboxes will be shown)
`\vbadness` (badness above which bad vboxes will be shown)
`\linepenalty` (amount added to badness of every line in a paragraph)
`\hyphenpenalty` (penalty for line break after discretionary hyphen)
`\exhyphenpenalty` (penalty for line break after explicit hyphen)
`\binoppenalty` (penalty for line break after binary operation)
`\relpenalty` (penalty for line break after math relation)
`\clubpenalty` (penalty for creating a club line at bottom of page)
`\widowpenalty` (penalty for creating a widow line at top of page)
`\displaywidowpenalty` (ditto, before a display)
`\brokenpenalty` (penalty for page break after a hyphenated line)
`\predisplaypenalty` (penalty for page break just before a display)
`\postdisplaypenalty` (penalty for page break just after a display)
`\interlinepenalty` (additional penalty for page break between lines)
`\floatingpenalty` (penalty for insertions that are split)
`\outputpenalty` (penalty at the current page break)
`\doublehyphendemerits` (demerits for consecutive broken lines)
`\finalhyphendemerits` (demerits for a penultimate broken line)
`\adjdemerits` (demerits for adjacent incompatible lines)

`\looseness` (change to the number of lines in a paragraph)
`\pausing` (positive if pausing after each line is read from a file)
`\holdinginserts` (positive if insertions remain dormant in output box)
`\tracingonline` (positive if showing diagnostic info on the terminal)
`\tracingmacros` (positive if showing macros as they are expanded)
`\tracingstats` (positive if showing statistics about memory usage)
`\tracingparagraphs` (positive if showing line-break calculations)
`\tracingpages` (positive if showing page-break calculations)
`\tracingoutput` (positive if showing boxes that are shipped out)
`\tracinglostchars` (positive if showing characters not in the font)
`\tracingcommands` (positive if showing commands before they are executed)
`\tracingrestores` (positive if showing deassignments when groups end)
`\language` (the current set of hyphenation rules)
`\uchyph` (positive if hyphenating words beginning with capital letters)
`\lefthyphenmin` (smallest fragment at beginning of hyphenated word)
`\righthyphenmin` (smallest fragment at end of hyphenated word)
`\globaldefs` (nonzero if overriding `\global` specifications)
`\defaulthyphenchar` (`\hyphenchar` value when a font is loaded)
`\defaultskewchar` (`\skewchar` value when a font is loaded)
`\escapechar` (escape character in the output of control sequence tokens)
`\endlinechar` (character placed at the right end of an input line)
`\newlinechar` (character that starts a new output line)
`\maxdeadcycles` (upper bound on `\deadcycles`)
`\hangafter` (hanging indentation changes after this many lines)
`\fam` (the current family number)
`\mag` (magnification ratio, times 1000)
`\delimiterfactor` (ratio for variable delimiters, times 1000)
`\time` (current time of day in minutes since midnight)
`\day` (current day of the month)
`\month` (current month of the year)
`\year` (current year of our Lord)
`\showboxbreadth` (maximum items per level when boxes are shown)
`\showboxdepth` (maximum level when boxes are shown)
`\errorcontextlines` (maximum extra context shown when errors occur)

这些参数的前几个值的单位是控制断行和分页的“badness”和“penalty”。接着是表示罚点的参数；罚点的单位实际上是“badness”的平方，因此这些参数的值相当要大一些。相反，再接下来的几个参数

(\looseness, \pausing, 等等)一般是非常小的值(-1, 0, 1 或 2)。剩下的是几个杂项参数。 TEX 在运行时要调用日期和时间, 只有在操作系统提供这些信息时才这样; 但是它不会持续记录这些: 用户可以象任意普通参数那样改变 \time。第十章指出, 在 TEX 已经指定一个特殊的放大率后, \mag 就不能再改变了。

⟨dimen parameter⟩ 是下列某个:

\hfuzz (maximum overrun before overfull hbox messages occur)
\vfuzz (maximum overrun before overfull vbox messages occur)
\overfullrule (width of rules appended to overfull boxes)
\emergencystretch (reduces badnesses on final pass of line-breaking)
\hsize (line width in horizontal mode)
\vsize (page height in vertical mode)
\maxdepth (maximum depth of boxes on main pages)
\splitmaxdepth (maximum depth of boxes on split pages)
\boxmaxdepth (maximum depth of boxes on explicit pages)
\lineskiplimit (threshold where \baselineskip changes to \lineskip)
\delimitershortfall (maximum space not covered by a delimiter)
\nulldelimiterspace (width of a null delimiter)
\scriptspace (extra space after subscript or superscript)
\mathsurround (kerning before and after math in text)
\predisplaysize (length of text preceding a display)
\displaywidth (length of line for displayed equation)
\displayindent (indentation of line for displayed equation)
\parindent (width of \indent)
\hangindent (amount of hanging indentation)
\hoffset (horizontal offset in \shipout)
\voffset (vertical offset in \shipout)

⟨glue parameter⟩ 可能是:

\baselineskip (desired glue between baselines)
\lineskip (interline glue if \baselineskip isn't feasible)
\parskip (extra glue just above paragraphs)
\abovedisplayskip (extra glue just above displays)
\abovedisplayshortskip (ditto, following short lines)
\belowdisplayskip (extra glue just below displays)
\belowdisplayshortskip (ditto, following short lines)
\leftskip (glue at left of justified lines)
\rightskip (glue at right of justified lines)

\topskip (glue at top of main pages)
\splittopskip (glue at top of split pages)
\tabskip (glue between aligned entries)
\spaceskip (glue between words, if nonzero)
\xspaceskip (glue between sentences, if nonzero)
\parfillskip (additional \rightskip at end of paragraphs)

最后, 有三个可用的 ⟨muglue parameter⟩ 记号:

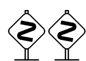
\thinmuskip (thin space in math formulas)
\medmuskip (medium space in math formulas)
\thickmuskip (thick space in math formulas)

所有这些量都已经在本书的某些地方讨论过了, 你可以从附录 I 查找其出处。

TEX 还有记号列参数。这样的参数不能出现在 ⟨number⟩ 等的定义中, 但是为了参数表的完整, 我们也把它们列出来。⟨token parameter⟩ 是任意的:

\output (the user's output routine)
\everypar (tokens to insert when a paragraph begins)
\everymath (tokens to insert when math in text begins)
\everydisplay (tokens to insert when display math begins)
\everyhbox (tokens to insert when an hbox begins)
\everyvbox (tokens to insert when a vbox begins)
\everyjob (tokens to insert when the job begins)
\everycr (tokens to insert after every \cr or nonredundant \crcr)
\errhelp (tokens that supplement an \errmessage)

所有这五类共有 103 个参数。


▶ **练习24.5**
当运行一个任务时, 看看 \everyjob 怎样才是非空的。

现在要回到我们最初的任务了, 即讨论 TEX 的消化过程所遇见的命令。很多命令在所有模式下都是一样的。大多数这种重要的命令都称为指定, 因为它们都给控制系列指定了新的意思, 或者给 TEX 内部量指定了新的值。例如, '\def\a{a}', '\parshape=1 5pt 100pt', '\advance\count20 by-1'和'\font\ff = cmff at 20pt'都是指定, 并且在所有模式下其结果都相同。指定命令通常包含一个等号 =, 但是在所有情况下这个等号是可有可无的; 如果你不在乎所得到的 TEX 代码看起来不象一个指定, 就可以扔掉这个等号。

⟨assignment⟩ ⟶ ⟨non-macro assignment⟩ | ⟨macro assignment⟩
⟨non-macro assignment⟩ ⟶ ⟨simple assignment⟩
  | \global⟨non-macro assignment⟩

⟨macro assignment⟩ ⟶ ⟨definition⟩ | ⟨prefix⟩⟨macro assignment⟩
⟨prefix⟩ ⟶ \global | \long | \outer
⟨equals⟩ ⟶ ⟨optional spaces⟩ | ⟨optional spaces⟩ $=_{12}$

这个语法表明, 每个指定前面可以有 \global, 但是只有在宏定义的指定前允许有 \long 或 \outer。顺便说一下, 如果在指定时参数 \globaldefs 是正值, 那么就意味着要自动加上前缀 \global; 但是如果在指定时 \globaldefs 是负值, 就忽略掉前缀 \global。如果 \globaldefs 是零(一般是这样的), 就按照是否是全局指定来确定 \global 出现与否。

⟨definition⟩ ⟶ ⟨def⟩⟨control sequence⟩⟨definition text⟩
⟨def⟩ ⟶ \def | \gdef | \edef | \xdef
⟨definition text⟩ ⟶ ⟨parameter text⟩⟨left brace⟩⟨balanced text⟩⟨right brace⟩

这里的控制系列表示一个控制系列记号或者是一个活动符记号; ⟨left brace⟩ 和 ⟨right brace⟩ 是类代码分别为 1 和 2 的显式字符记号。⟨parameter text⟩ 不包括记号 ⟨left brace⟩ 或 ⟨right brace⟩, 并且遵守第二十章的规则。所有出现在 ⟨balanced text⟩ 中的 ⟨left brace⟩ 和 ⟨right brace⟩ 记号必须象括号那样正确嵌套。命令 \gdef 等价于 \global\def, \xdef 等价于 \global\edef。 TeX 读入 ⟨control sequence⟩ 和 ⟨parameter text⟩ 记号以及开符号 ⟨left brace⟩ 而不展开它们; 只有在 \edef 和 \xdef 的情况下, 它从展开记号 ⟨balanced text⟩⟨right brace⟩。

我们下面要讨论的几个命令的语法有点象宏定义, 但是 ⟨parameter text⟩ 被任意个空格或'\relax'所代替, 并且记号 ⟨left brace⟩ 可以是隐含的:

⟨filler⟩ ⟶ ⟨optional spaces⟩ | ⟨filler⟩\relax⟨optional spaces⟩
⟨general text⟩ ⟶ ⟨filler⟩{⟨balanced text⟩⟨right brace⟩

⟨general text⟩ 的主要用处是给出其中的 ⟨balanced text⟩。

可能有很多种不同的指定, 但是它们几乎都属于几个模式, 就象下面的语法规则表明的那样:

⟨simple assignment⟩ ⟶ ⟨variable assignment⟩ | ⟨arithmetic⟩
  | ⟨code assignment⟩ | ⟨let assignment⟩ | ⟨shorthand definition⟩
  | ⟨fontdef token⟩ | ⟨family assignment⟩ | ⟨shape assignment⟩
  | \read⟨number⟩ to ⟨optional spaces⟩⟨control sequence⟩
  | \setbox⟨8-bit number⟩⟨equals⟩⟨filler⟩⟨box⟩
  | \font⟨control sequence⟩⟨equals⟩⟨file name⟩⟨at clause⟩
  | ⟨global assignment⟩
⟨variable assignment⟩ ⟶ ⟨integer variable⟩⟨equals⟩⟨number⟩
  | ⟨dimen variable⟩⟨equals⟩⟨dimen⟩
  | ⟨glue variable⟩⟨equals⟩⟨glue⟩
  | ⟨muglue variable⟩⟨equals⟩⟨muglue⟩
  | ⟨token variable⟩⟨equals⟩⟨general text⟩
  | ⟨token variable⟩⟨equals⟩⟨filler⟩⟨token variable⟩

⟨arithmetic⟩ ⟶ `\advance`⟨integer variable⟩⟨optional `by`⟩⟨number⟩

  | `\advance`⟨dimen variable⟩⟨optional `by`⟩⟨dimen⟩

  | `\advance`⟨glue variable⟩⟨optional `by`⟩⟨glue⟩

  | `\advance`⟨muglue variable⟩⟨optional `by`⟩⟨muglue⟩

  | `\multiply`⟨numeric variable⟩⟨optional `by`⟩⟨number⟩

  | `\divide`⟨numeric variable⟩⟨optional `by`⟩⟨number⟩

⟨optional `by`⟩ ⟶ `by` | ⟨optional spaces⟩

⟨integer variable⟩ ⟶ ⟨integer parameter⟩ | ⟨countdef token⟩

  | `\count`⟨8-bit number⟩

⟨dimen variable⟩ ⟶ ⟨dimen parameter⟩ | ⟨dimendef token⟩

  | `\dimen`⟨8-bit number⟩

⟨glue variable⟩ ⟶ ⟨glue parameter⟩ | ⟨skipdef token⟩

  | `\skip`⟨8-bit number⟩

⟨muglue variable⟩ ⟶ ⟨muglue parameter⟩ | ⟨muskipdef token⟩

  | `\muskip`⟨8-bit number⟩

⟨token variable⟩ ⟶ ⟨token parameter⟩ | ⟨toksdef token⟩

  | `\toks`⟨8-bit number⟩

⟨numeric variable⟩ ⟶ ⟨integer variable⟩ | ⟨dimen variable⟩

  | ⟨glue variable⟩ | ⟨muglue variable⟩

⟨code assignment⟩ ⟶ ⟨code name⟩⟨8-bit number⟩⟨equals⟩⟨number⟩

⟨let assignment⟩ ⟶ `\futurelet`⟨control sequence⟩⟨token⟩⟨token⟩

  | `\let`⟨control sequence⟩⟨equals⟩⟨one optional space⟩⟨token⟩

⟨shorthand definition⟩ ⟶ `\chardef`⟨control sequence⟩⟨equals⟩⟨8-bit number⟩

  | `\mathchardef`⟨control sequence⟩⟨equals⟩⟨15-bit number⟩

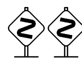
  | ⟨registerdef⟩⟨control sequence⟩⟨equals⟩⟨8-bit number⟩

⟨registerdef⟩ ⟶ `\countdef` | `\dimendef` | `\skipdef` | `\muskipdef` | `\toksdef`

⟨family assignment⟩ ⟶ ⟨family member⟩⟨equals⟩⟨font⟩

⟨shape assignment⟩ ⟶ `\parshape`⟨equals⟩⟨number⟩⟨shape dimensions⟩

在 ⟨code assignment⟩ 结尾处的 ⟨number⟩ 必须是非负的, 除了要指定的是 `\delcode` 外。还有, 对 `\catcode` 此数至多为 15, 对 `\mathcode` 至多为 32768, 对 `\lccode` 或 `\uccode` 至多为 255, 对 `\sfcode` 至多为 32767, 对 `\delcode` 至多为 $2^{24}-1$。在 ⟨number⟩ 为 n 的 ⟨shape assignment⟩ 中, 如果 $n \le 0$, 那么 ⟨shape assignment⟩ 为 ⟨empty⟩, 否则它们由 $2n$ 个连续出现的 ⟨dimen⟩ 组成。当 TeX 遇见 `\let` 和 `\futurelet` 的变量时, 不会把记号展开。



▶ **练习24.6**

在本章的前面部分, 我们讨论了显式和隐式字符记号的区别。看看怎样利用 (a) `\futurelet`, (b) `\let`, 把控制系列 `\cs` 变成隐式的。

到现在讨论的所有指定都遵守 TEX 的编组结构; 即, 当当前组结束时, 被修改的量都要恢复其以前的值, 除非修改是全局的。其余的指定是不同的, 因为它们控制的是 TEX 的全局字体表或连字表, 或者它们控制的是某些无法编组的内部性质的变量。在下列所有情况下, `\global` 是否作为前缀出现无关紧要。

⟨global assignment⟩ ⟶ ⟨font assignment⟩
 | ⟨hyphenation assignment⟩
 | ⟨box size assignment⟩
 | ⟨interaction mode assignment⟩
 | ⟨intimate assignment⟩
⟨font assignment⟩ ⟶ `\fontdimen`⟨number⟩⟨font⟩⟨equals⟩⟨dimen⟩
 | `\hyphenchar`⟨font⟩⟨equals⟩⟨number⟩
 | `\skewchar`⟨font⟩⟨equals⟩⟨number⟩
⟨at clause⟩ ⟶ `at` ⟨dimen⟩ | `scaled` ⟨number⟩ | ⟨optional spaces⟩
⟨hyphenation assignment⟩ ⟶ `\hyphenation`⟨general text⟩
 | `\patterns`⟨general text⟩
⟨box size assignment⟩ ⟶ ⟨box dimension⟩⟨8-bit number⟩⟨equals⟩⟨dimen⟩
⟨interaction mode assignment⟩ ⟶ `\errorstopmode` | `\scrollmode`
 | `\nonstopmode` | `\batchmode`
⟨intimate assignment⟩ ⟶ ⟨special integer⟩⟨equals⟩⟨number⟩
 | ⟨special dimen⟩⟨equals⟩⟨dimen⟩

当指定了 `\fontdimen` 的值时, ⟨number⟩ 必须是正值, 并且不大于字体度量文件中参数的数值, 除非此字体的信息刚刚载入 TEX 的内存; 在后一种情况下, 允许增大参数的数值(见附录 F)。当讨论内部整数或尺寸时, 上面的量 ⟨special integer⟩ 和 ⟨special dimen⟩ 要列出。当 `\prevgraf` 被设定为一个 ⟨number⟩ 时, 此数必须是非负的。

⟨file name⟩ 的语法在 TEX 中没有标准, 因为不同的操作系统有不同的约定。你应该向本局域系统的高手请教, 看看怎样给出执行文件的名称。但是, 下面的原理是普遍成立的: (在展开后)⟨file name⟩ 由 ⟨optional spaces⟩ 后面跟显式字符记号而组成。在所有的 TEX 所在系统中, 六个以下普通字母和/或数字加一个空格而得到的文件名应该基本上都是一样的。在文件名中, 大写字母与相应的小写字母可能不等价; 例如, 如果你用到字体 `\cmr10` 和 `CMR10`, TEX 不会认为它们是类似的, 虽然这两种字体所用的字体度量文件可能是同一个。

对象‘`\chardef\cs=10\cs`’和‘`\font\cs=name\cs`’这样的结构, TEX 有预防措施, 直到指定结束后, 它才展开第二个 `\cs`。

我们关于指定的讨论只剩下 `\setbox` 了, 它包括一个叫做 ⟨box⟩ 的量, 而我们还未定义它。其语法如下:

⟨box⟩ ⟶ `\box`⟨8-bit number⟩ | `\copy`⟨8-bit number⟩
 | `\lastbox` | `\vsplit`⟨8-bit number⟩ `to` ⟨dimen⟩
 | `\hbox`⟨box specification⟩`{`⟨horizontal mode material⟩`}`

| \vbox⟨box specification⟩{⟨vertical mode material⟩}

| \vtop⟨box specification⟩{⟨vertical mode material⟩}

⟨box specification⟩ ⟶ to ⟨dimen⟩⟨filler⟩

| spread ⟨dimen⟩⟨filler⟩ | ⟨filler⟩

对象 \lastbox 不允许出现在数学模式中, 当主垂直列完全送到当前页面时, 也不允许出现在垂直模式中。但是它可以出现在水平模式和内部垂直模式中; 在这样的模式下, 如果最后的项目是一个 hbox 或 vbox, 那么它指向(并且去掉)当前列的最后的项目。

⟨box⟩ 的最后三个选项给出了一种新情况: 在一个 \hbox 中的 ⟨horizontal mode material⟩ 和在一个 ⟨vbox⟩ 中的 ⟨vertical mode material⟩ 不能在象 ⟨8-bit number⟩ 或 ⟨dimen⟩ 那样的一个命令中直接消化掉;

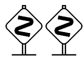
下面是实际发生的情况: 对象

\setbox⟨number⟩=\hbox to⟨dimen⟩{⟨horizontal mode material⟩}

这样一个命令, TEX 要计算 ⟨number⟩ 和 ⟨dimen⟩, 并且为了安全把这些值放在一个"堆栈"上。接着 TEX 读入'{'(它表示前面讨论的显式或隐式组开始符号), 并且开始另一层编组。这时, TEX 进入受限水平模式并且在此模式下执行命令。现在, 可以构造任意复杂的盒子; 当构造盒子时, 此盒子最后指定给一个 \setbox 命令, 这样就不会影响 TEX 的工作。最后, 当相匹配的'}'出现时, TEX 就把刚刚结束的盒子中由指定改变的值复原; 接着把 hbox 打包(使用存放在堆栈中的尺寸), 并且完成命令 \setbox, 回到 \setbox 时所处的模式。

现在我们要讨论其它命令, 它们象指定这样, 在任何模式下都得到基本相同的结果。

- \relax. 这个很简单: TEX 什么也不做。
- }. 这个有点难, 因为它依赖于当前组。现在 TEX 应该处在以 { 开始的一个组中; 并且它知道此组开始的理由。因此它给出相应的结束, 恢复非全局指定的设置, 并且离开这个组。这时, TEX 可能离开当前模式, 并且返回前面一个还起作用的模式。
- \begingroup. 当 TEX 遇见这个命令时, 就进入一个组, 这个组必须以 \endgroup 结束, 而不是 }。模式却不会改变。
- \endgroup. 目前 TEX 应该正在处理以 \begingroup 开始的一个组。在此组中非全局指定修改的量将恢复为其原来的值。 TEX 离开这个组, 但是保持模式不变。
- \show⟨token⟩, \showbox⟨8-bit number⟩, \showlists, \showthe⟨internal quantity⟩. 这些命令帮助你了解 TEX 正在干什么。跟在 \showthe 后面的记号是可以跟在 \the 后面的任何对象, 见第二十章的讨论。



▶ **练习24.7**

复习一下第二十章中可以跟在 \the 后面的这些规则, 并且用一种与已经讨论过的其它语法规则相类似的方法, 构造出定义 ⟨internal quantity⟩ 的正式语法。

- \shipout⟨box⟩. 在 ⟨box⟩ 生成后——可能是显式地构造出它并且在构造时要改变模式, 就象前面 \hbox 的讨论那样——其内容要送到 \dvi 文件(见第二十三章)。

▪ \ignorespaces⟨optional spaces⟩. TEX 一直读入到遇见一个 ⟨space token⟩ 后, 就读入(并且展开)记号,

▪ \afterassignment⟨token⟩. 这个 ⟨token⟩ 被保存在一个特殊的地方; 在下一个指定命令刚刚执行完毕后, 再把它插回到输入中。后面不必直接跟一个指定; 如果在下一个指定前另一个 \afterassignment 被执行了, 那么第二个就不用管第一个。如果下一个指定是一个 \setbox, 并且被指定的 ⟨box⟩ 是 \hbox, \vbox 或 \vtop, 那么在盒子的构造中, ⟨token⟩ 就被插入到 { 紧后面, 而不是 } 后面; 它还出现在由 \everyhbox 或 \everyvbox 所插入任意记号的紧前面。

▪ \aftergroup⟨token⟩. 这个 ⟨token⟩ 被保存在 TEX 的堆栈上; 在当前组结束及其局域指定都解除紧后面, 它将被插回到输入。如果在同一组中出现几个 \aftergroup 命令, 那么相应的命令按照同样的次序来读入; 例如, '{\aftergroup\a\aftergroup\b}'得到的是'\a\b'。

▪ \uppercase⟨general text⟩, \lowercase⟨general text⟩. 在通用文本中的 ⟨balanced text⟩ 通过利用 \uccode 或 \lccode 表转换为大写或小写格式, 见第七章的讨论; 不进行展开。 因此, TEX 会再次读入此 ⟨balanced text⟩。

▪ \message⟨general text⟩, \errmessage⟨general text⟩. 所得到的文本(转换后)被输出到用户的终端, 如果是 \errmessage 就使用错误信息的格式。在后一种情况下, 记号 \errhelp 在非空并且用户要求的情况下会显示出来。

▪ \openin⟨4-bit number⟩⟨equals⟩⟨filename⟩, \closein⟨4-bit number⟩. 这些命令打开或关闭所给定的输入流, 这些输入流为第二十章中讨论的 \read 的指定所使用。

▪ \immediate\openout⟨4-bit number⟩⟨equals⟩⟨filename⟩, \immediate\closeout⟨4-bit number⟩. 给定的输出流被打开或关闭, 为命令 \write 所用, 见第二十一章的讨论。

▪ \immediate\write⟨number⟩⟨general text⟩. 所得到的文本被写入到对应于给定流编号的文件中, 如果这样一个文件是打开的话。否则, 就写入到用户终端和 log 文件中。(见第二十一章; 如果 ⟨number⟩ 是负的, 就不输出到终端。)

上面就是不依赖于模式的所有命令的列表, 即, 不直接影响 TEX 所构建的列的命令。当 TEX 处在垂直模式或内部垂直模式中时, 它就在构建一个垂直列; 当 TEX 处在水平模式或受限水平模式中时, 它就在构建一个水平列; 当 TEX 处在数学模式或列表数学模式下时, 可以猜出, 它在构建一个数学列。对每个情况下, 我们都可以讨论"当前列"; 并且有一些命令在所有模式下结果是相同的, 除非它们处理的是不同种类的列:

▪ \openout⟨4-bit number⟩⟨equals⟩⟨filename⟩, \closeout⟨4-bit number⟩, \write⟨number⟩⟨general text⟩. 这些命令记录了追加到当前列的一个"无名"项目。此命令在 \shipout 应用到这个列期间后才执行, 除非此列是引导符中盒子的内容。

▪ \special⟨general text⟩. 所得到的文本被展开并且放在一个"无名"项目中, 它要追加到当前列。此文本最后作为后续软件的指令而出现在 dvi 文件中(见第二十一章)。

▪ \penalty⟨number⟩. 给定数值的 penalty 项目要追加到当前列。在垂直模式下, TEX 还要进行页面构建(见下面)。

- \kern⟨dimen⟩, \mkern⟨mudimen⟩. 给定尺寸的 kern 项目要追加到当前列。在垂直模式下, 它表示垂直空白; 否则它表示的是水平空白。\mkern 只允许出现在数学模式中。

- \unpenalty, \unkern, \unskip. 如果当前列上的最后一个项目的类型分别是 penalty, kern 或者粘连(可能含有引导符), 那么此项目将从当前列中删除。但是, 如果此时的主垂直列已经完全送到当前页面, 那么象 \lastbox 这些命令不允许出现在垂直模式中, 因为 TEX 从不从当前页面删除项目。

- \mark⟨general text⟩. 所得到的文本被展开, 并且放到一个标记项目中, 此项目要追加到当前列。如果这个标记项目曾经进入过垂直列, 那么此文本最后会变成 \topmark, \firstmark, \botmark, \splitfirstmark 和/或 \splitbotmark 的替换文本。(标记项目可以出现在水平列和数学列, 但是直到它们从这些列中"转移"出来才能派上用场。下面和第二十五章讨论了这个转移过程。)

- \insert⟨8-bit number⟩⟨filler⟩{⟨vertical mode material⟩}; 此 ⟨8-bit number⟩ 不能是 255。这个'{'使得 TEX 进入内部垂直模式和一个新层次的编组。当遇到相匹配的'}'时, 此垂直列被放在一个插入项目中, 并且利用组刚刚结束时的\splittopskip, \splitmaxdepth 和 \floatingpenalty 的当前值追加到当前列。(见第十五章。) 只有在这个插入项目出现在 TEX 的主垂直列中时, 它才最终变成页面插入对象, 因此如果它原来在垂直列或数学列中, 就必须"转移"出来。在把一个 \insert 追加到垂直模式中后, TEX 还要进行页面构建(见下面)。

- \vadjust⟨filler⟩{⟨vertical mode material⟩}. 它类似于 \insert; 所构造的垂直列放在当前的调整项目中, 并且追加到当前列。但是, \vadjust 不允许出现在垂直模式中。当一个调整项目从水平列转移到垂直列时, 在调整项目中的垂直列被"打开", 并且直接放在此封装列中。

* * *

在本章, 到现在为止讨论的几乎所有对象都同样出现在"水平模式汇总"和"数学模式汇总"这每一章中, 因为这些命令在所有模式下的结果都是一样的。第二十五和二十六章比本章短很多, 因为它不必重复这些与模式无关的对象。

但是现在无名要讨论与模式有关的命令; 在本章最后, 我们讨论在垂直模式和内部垂直模式中其余的命令。

垂直模式的一个特有的东西就是在第十五章中讨论的页面构建。TEX 定期从主垂直列中取出内容, 并且把它从"备选列"送到"当前页面"。此时可能要调用输出例行程序。只要 TEX 正在从当前备选列中送出内容, 我们就称它在*进行页面构建*。备选列这个名称只在最外层垂直列中出现, 因此当在内部垂直模式下进行页面构建时什么也不做。

另一个垂直模式所特有的东西是行间粘连, 它插入到盒子前面, 使用的是第十二章中讨论的 \prevdepth, \baselineskip, \lineskip 和 \lineskiplimit 的值。如果用一个命令把 \prevdepth 改变了, 那么就出现了下面特别描述的结果。只要 TEX 开始构建一个垂直列, \prevdepth 就最先设置为 -1000 pt, 这是来自于行间粘连的一个特殊值; 但是外层列的行间粘连设置延续到内层时 \halign 和 \noalign 的情况除外。

- \vskip⟨glue⟩, \vfil, \vfill, \vss, \vfilneg. 把一个粘连项目追加到当前垂直列。

■ ⟨leaders⟩⟨box or rule⟩⟨vertical skip⟩. 在这里, ⟨vertical skip⟩ 指的是刚刚提到的五个追加粘连的命令。⟨leaders⟩ 和 ⟨box or rule⟩ 的正式语法为:

⟨leaders⟩ ⟶ `\leaders` | `\cleaders` | `\xleaders`
⟨box or rule⟩ ⟶ ⟨box⟩ | ⟨vertical rule⟩ | ⟨horizontal rule⟩
⟨vertical rule⟩ ⟶ `\vrule`⟨rule specification⟩
⟨horizontal rule⟩ ⟶ `\hrule`⟨rule specification⟩
⟨rule specification⟩ ⟶ ⟨optional spaces⟩ | ⟨rule dimension⟩⟨rule specification⟩
⟨rule dimension⟩ ⟶ `width` ⟨dimen⟩ | `height` ⟨dimen⟩ | `depth` ⟨dimen⟩

生成引导符的粘连项目被追加到当前列。

■ ⟨space token⟩. 空格在垂直列中不起作用。

■ ⟨box⟩. 构造这个盒子, 并且如果所得到的是空的话就什么也没有得到。否则, 当前垂直列就得到了 (1). 行间粘连, 后面跟着 (2). 这个新盒子, 后面再跟着 (3). 从新盒子转移出来的垂直内容(当这个 ⟨box⟩ 是一个 `\hbox` 命令时)。接着, `\prevdepth` 被设定为这个新盒子的深度, 并且 TeX 进行页面构建。

■ `\moveleft`⟨dimen⟩⟨box⟩, `\moveright`⟨dimen⟩⟨box⟩. 它就象一个普通 ⟨box⟩ 命令一样, 但是追加到垂直列的这个新盒子还要向左或向右平移给定的距离。

■ `\unvbox`⟨8-bit number⟩, `\unvcopy`⟨8-bit number⟩. 如果所给定的盒子寄存器是置空的, 那么就什么也不做。否则, 寄存器必须包含一个 vbox。在此盒子中的垂直列不做任何改变地追加到当前列。`\prevdepth` 的值也不受影响。在 `\unvbox` 后此盒子寄存器变成置空的, 但是在 `\unvcopy` 后它保持不变。

■ ⟨horizontal rule⟩. 这个给定的标尺被追加到当前列。接着把 `\prevdepth` 设定为 -1000 pt; 当下一个盒子追加到本列时不能有行间粘连。

■ `\halign`⟨box specification⟩`{`⟨alignment material⟩`}`. 这个 ⟨alignment material⟩ 由导言和其后的零个或多个对齐的行组成; 见第二十二章。 TeX 进入一个由'`{`'和'`}`'表示的新层次的编组, 在其中对 `\tabskip` 的修改会受到限制。对齐的内容还可以出现行间的'`\noalign`⟨filler⟩`{`⟨vertical mode material⟩`}`'; 它会增加另一个层次的编组。当 TeX 处理 `\noalign` 组中的内容和追加对齐的行时, TeX 处在内部垂直模式中; 在对齐内容结束后, 所得到的内部垂直列将追加到封装的垂直列, 并且进行页面构建。在 `\halign` 时, `\prevdepth` 的值在内部垂直列的开头要用到, 当本对齐结束时, `\prevdepth` 最后的值会送到封装的垂直列, 这样在对齐开头和结尾处就正确计算出了行间粘连。当 TeX 处理对齐的各个单元时还要进入另一个层次的编组, 这时它处在受限水平模式中; 作为最后对齐的一部分, 各个单元都放在 hbox 中, 并且其垂直内容将转移到封装的垂直列。命令 `\noalign`, `\omit`, `\span`, `\cr`, `\crcr` 和 `&` (其中 `&` 表示类代码为 4 的显式或隐式字符)在去 TeX 的胃的过程中被对齐过程所截获, 这样它们就不作为命令而出现在胃中, 除非 TeX 不知道它们属于哪个对齐。

■ `\indent`. 这个 `\parskip` 粘连要追加到对齐列, 除非 TeX 处在内部垂直模式, 并且当前列是空的。接着 TeX 进入非受限的水平模式, 以宽度为 `\parindent` 的一个空 hbox 开始这个水平列。记号 `\everypar` 被插入到 TeX 的输入中。页面开始构建。当本段落最后结束时, 水平模式将出现如第二十五章讨论的结尾。

- \noindent. 它与 \indent 一样, 只是 TEX 以一个空列开始而不是以缩进开始。

- \par. 当 TEX 处在垂直模式中时, 原始命令 \par 不起作用, 但是如果在备选列中出现了一些内容, 并且段落的形状是知道的, 那么要进行页面构建。

- {. 它是类代码为 1 的字符记号或者是被 \let 等于这样一个字符记号的象 bgroup 这样的一个控制系列, 它使得 TEX 开始一个新层次的编组。当这样的组以'}'结束时, TEX 将撤消所有非全局的指定, 并且保持此时的模式不变。

- 某些命令与垂直模式不相容, 因为它们在本质上是水平模式的。当下列命令出现在垂直模式中时, 会使得 TEX 开始一个新段落:

⟨horizontal command⟩ ⟶ ⟨letter⟩ | ⟨otherchar⟩ | \char | ⟨chardef token⟩
  | \noboundary | \unhbox | \unhcopy | \valign | \vrule
  | \hskip | \hfil | \hfill | \hss | \hfilneg
  | \accent | \discretionary | \- | \␣ | $

这里的 ⟨letter⟩ 和 ⟨otherchar⟩ 表示类代码为 11 和 12 的显式或隐式字符记号。如果这些命令的任一个在垂直模式或内部垂直模式下作为命令而出现, 那么 TEX 自动执行象上面讨论的一样的 \indent 命令。这就进入水平模式, 在输入中有记号 \everypar, 其后 TEX 将再次遇见 ⟨horizontal command⟩。

- \end. 此命令不允许出现在内部垂直模式中。在正常的垂直模式下, 如果垂直列是空的并且 \deadcycles=0, 它将终止 TEX。否则, TEX 将把命令 \end 备份下来, 这样它可以再次读入; 接着, 在添加了输出例行程序的盒子/粘连/penalty 组合后, 它进行页面构建。(见第二十三章结尾。)

- \dump. (只允许出现在 INITEX 中, 而不允许在 TEX 的用户版中。) 这个命令就象 \end 一样, 但是它不能出现在组中。它输出一个格式文件, 此文件可以相当高速地载入到 TEX 的内存中以恢复当前状态。

- 除了上面的以外: 如果 TEX 任何其它原始命令出现在垂直模式中, 那么就给出一个错误信息, 并且 TEX 将试着合理地修复它。例如, 如果出现了上标或下标返回, 或者其它特有的数学命令出现了, 那么 TEX 就试着插入'$'(它将开始一个新段落并且进入数学模式)。另一方面, 如果象 \endcsname, \omit, \eqno 或 # 这些完全不该出现的记号出现在垂直模式中, TEX 就在给出错误后直接忽略掉它们。你可以试着输入一些完全愚蠢的内容, 看看会出现什么结果。(象第二十七章那样, 首先声明'\tracingall'以得到最多的信息。)

25. 水平模式汇总

继续讨论第二十四章开始的汇总, 现在, 我们讨论在水平模式和受限水平模式下 TEX 构建列时 TEX 的消化过程可以做的事情。

* * *

刚刚这三个星号可以在第二十四章快结束时找见。在三个星号前面的那章所有的内容都即适用于水平模式, 也适用于垂直模式, 因此我们就毋需除非那样的规则了。特别是, 第二十四章讨论了命令指定, 并且讨论了 kern, penalty, 标记, 插入对象, 调整和“无名”是怎样放在水平列中的。我们现在的目的是讨论水平模式特别的命令, 也就是它们在水平模式下与在垂直和数学模式下的结果不同。 水平模式一个特有的东西是“间距因子”, 它象第十二章讨论的那样来修改间距的宽度。如果要用命令改变 `\spacefactor` 的值, 那么就要特别注意一下。当 TEX 开始形成水平列时, 间距因子最初设定为 1000, 但是在 `\valign` 和 `\noalign` 时, 外层列的间距因子会延续到内层来。

- `\hskip`⟨glue⟩, `\hfil`, `\hfill`, `\hss`, `\hfilneg`. 把一个粘连项目追加到当前列。

- ⟨leaders⟩⟨box or rule⟩⟨horizontal skip⟩. 这里的 ⟨horizontal skip⟩ 指的是刚刚讨论的五个追加粘连的命令; ⟨leaders⟩ 和 ⟨box or rule⟩ 的正式语法见第二十四章。生成引导符的粘连项目要追加到当前列。

- ⟨space token⟩. 空白把粘连追加到当前列; 粘连的具体量与 `\spacefactor`, 当前字体, `\spaceskip` 和 `\xspaceskip` 有关, 见第十二章的讨论。

- `\␣`. 控制空格命令把粘连追加到当前列, 其量与间距因子为 1000 时 ⟨space token⟩ 插入的量相同。

- ⟨box⟩. 构造这个盒子, 并且如果所得到的是置空的就什么也不会出现。否则, 把新盒子追加到当前列, 并且把间距因子设置为 1000。

- `\raise`⟨dimen⟩⟨box⟩, `\lower`⟨dimen⟩⟨box⟩. 它与普通的 ⟨box⟩ 命令是一样的, 但是追加到水平列的新盒子还要向上或向下平移所给定的量。

- `\unhbox`⟨8-bit number⟩, `\unhcopy`⟨8-bit number⟩. 如果给定的盒子寄存器是置空的, 那么什么也不出现。否则, 寄存器必须包含一个 hbox。在此盒子中的水平列要追加到当前水平列, 而不做任何改变。`\spacefactor` 的值不受影响。在 `\unhbox` 后此盒子寄存器变成置空的, 在 `\unhcopy` 后它保持不变。

- ⟨vertical rule⟩. 把给定的标尺追加到当前列, 并且把 `\spacefactor` 设置为 1000。

- `\valign`⟨box specification⟩`{`⟨alignment material⟩`}`. ⟨alignment material⟩ 由导言后面跟零个或更多个要对齐的栏组成; 见第二十二章。 TEX 进入由‘`{`’和‘`}`’表示的一个新层次的编组, 其中对 `\tabskip` 的修改是受到限制的。对齐的内容还可以包含栏之间的‘`\noalign`⟨filler⟩`{`⟨horizontal mode material⟩`}`’; 这增加了另一个层次的编组。当处理 `\noalign` 组中的内容并且当追加对齐的栏时, TEX 处于受限水平模式中; 在对齐结束后, 所得到的不变水平列将追加到封装的水平列。在 `\valign` 中 `\spacefactor` 的值使用的是内部水平列开始时的值, 当对齐结束后, `\spacefactor` 最后的值要送到封装的水平列。在每栏结束后, 间距因子就设置为 1000; 因此它只影响在 `\noalign` 组中的结果。当处理对齐的每个栏时, TEX 还要进入更深层的编组, 此时它处在内部垂直模式; 各个单元都是作为最后对齐的一部分而放在 vbox 中。

- \indent. 把宽度为 \parindent 的空盒子追加到当前列, 并且把间距因子设置为 1000。

- \noindent. 此命令在水平模式下没有什么作用。

- \par. 原始命令 \par, 在 plain TeX 中也称为 \endgraf, 它在受限水平模式下什么也不做。但是它终止当前水平模式: 当前列通过 命令 \unskip \penalty10000 \hskip\parfillskip 来完成, 接着, 它如第十四章讨论的那样分段为行, 并且 TeX 回到封装的垂直或内部垂直模式。把段落的行追加到封装的垂直列, 其间插入行间粘连和行间 penalty, 并且把水平列中的垂直内容转移出来。接着 TeX 进行页面构建。

- {. 类代码为 1 的字符记号, 或者被 \let 等于这样的字符记号的象 \bgroup 这样的控制系列, 它使得 TeX 开始一个新层次的编组。当这样的组用'}'结束后, TeX 撤消所有的非全局指定, 并且保持其目前的模式不变。

- 某些命令与水平模式不相容, 因为它们在本质上是垂直模式的。当下列命令出现在非受限水平模式中时, 会使 TeX 结束当前段落:

  ⟨vertical command⟩ ⟶ \unvbox | \unvcopy | \halign | \hrule
  | \vskip | \vfil | \vfill | \vss | \vfilneg | \end | \dump

在受限水平模式下, 不允许出现一个 ⟨vertical command⟩, 但是在正常水平模式下, 这会使得 TeX 把记号 par 插入到输入中; 在读入和展开这个 par 后, TeX 就再次遇见了 ⟨vertical command⟩ 记号。(要用到控制系列 \par 当前的意思; par 可能不再表示 TeX 的原始命令 \par。)

- ⟨letter⟩, ⟨otherchar⟩, \char⟨8-bit number⟩, ⟨chardef token⟩, \noboundary. 所有最常用的命令是字符命令, 它把一个当前字体的字符追加到当前水平列。如果两个或多个这种命令连续出现, TeX 就把它们看作一个整体, 按照字体中的信息把它们连字符化和/或插入间距调整。(如果 \noboundary 不出现的话, 连字符化和间距调整可能会受字符左右两边看不见的"边界"的影响。) 每个字符命令都调整 \spacefactor, 使用的是第十二章讨论的 \sfcode 表。在非受限水平模式中, 要把'\discretionary{}{}{}'项目追加到代码为本字体的 \hyphenchar 的字符后面, 或者是以这样一个字符结尾的一个序列形成的连字符后面。

- \accent⟨8-bit number⟩⟨optional assignments⟩. 这里的 ⟨optional assignments⟩ 表示零个或多个 ⟨assignment⟩ 命令。如果指定后面没有一个 ⟨character⟩, 其中 ⟨character⟩ 表示前一段刚刚讨论的任何命令, 那么 TeX 把 \accent 当成 \char 来处理, 但是间距因子为 1000。否则, 跟在指定后面的字符就被对应于 ⟨8-bit number⟩ 的字符加上重音。(插入指定的目的是允许所加的重音字符和被加重音的字符可以是不同的字体。) 如果这个重音必须向上或向下移动, 那么就把它放在一个 hbox 中来升降。这样重音就通过调整间距的方式叠放在字符上, 按照这种方法, 重音的宽度不会影响所得到的水平列的宽度。最后, TeX 设置 \spacefactor=1000。

- \/. 如果当前列上的最后一个项目是字符或连字符, 那么其 italic 校正所需的间距调整要追加上。

- \discretionary⟨general text⟩⟨general text⟩⟨general text⟩. 这三个通用文本在受限水平模式下来处理。它们只包含宽度固定的内容; 因此, 在这种情况下它们不真那么通用。更准确地说, 由任意可断点的通用文本形成的水平列只能包含字符, 连字符, 间距调整, 盒子和标尺; 不应该有粘连或 penalty 项目等。这个命令把一个任意可断点项目追加到当前列; 见第十四章任意可断点的含义的讨论。它不改变间距因子。

- `\-`. 这个“任意连字符”命令在附录 H 中给出定义。

- `\setlanguage`⟨number⟩. 见附录 H 的结果。

- `$`. “数学转换”符使 TEX 按下列方式进入数学模式或列表数学模式: TEX 遇见下列记号而不展开它们。如果此记号是一个 `$` 并且 TEX 目前处在非受限水平模式中, 那么 TEX 象上面讨论的那样把当前段落分为行(除非当前列是空的), 再返回封装的垂直模式或内部垂直模式, 计算象 `\prevdepth`, `\displaywidth` 和 `\predisplaysize` 的值, 进入新一层次的编组, 把记号 `\everydisplay` 插入到输入中, 进行页面构建, 在列表数学模式下处理‘⟨math mode material⟩`$$`’, 象第十九章讨论的那样把此列表公式放在封装的垂直列中(把垂直内容转移处理), 再次进行页面构建, 把 `\prevgraf` 增加 3, 并且再回到水平模式, 开始一个空列并且设置间距因子为 1000。(你看懂了吗”) 否则, TEX 就把所读入的记号放回输入, 进入一个新层次的编组, 插入记号 `\everymath`, 并且处理‘⟨math mode material⟩`$`’; 这个数学模式的内容被转换为一个水平列并且追加到当前列, 两边放上“math-on”项目和“math-off”项目, 并且把间距因子设置为 1000。这些规则的一个后果就是, 在受限水平模式下的‘`$$`’直接得到一个空数学公式。

- 除上面的以外: 如果任何其它 TEX 的原始命令出现在水平模式中, 那么就给出错误信息, 并且 TEX 试着按照合理的方式修复它。例如, 如果出现了上标或下标, 或者出现了其它数学特有的命令, 那么 TEX 就试着在这个讨厌的记号紧前面插入一个‘`$`’; 它就进入了数学模式。

26. 数学模式汇总

为了完成在第二十四章开始的汇总, 我们来讨论在数学模式或列表数学模式下 TEX 构建列时 TEX 所用到的命令。

* * *

刚刚出现的三个星号可以在第二十四章结尾附近找到。在那章的三个星号前面出现的所有内容对数学模式和垂直模式都是一样的, 因此我们毋需重复所有这些规则。特别是, 第二十四章讨论了指定命令, 并且给出了间距调整, penalty, 边界, 插入对象, 调整和"无名"是怎样放到数学列中的。我们现在的目的是讨论数学模式固有的命令, 在这个意义上它们在数学模式下与在垂直或水平模式下的性质不同。

数学列与 TEX 的其它列有点不同, 因为它们包含三叉"原子"(见第十七章)。原子分为十三类: Ord, Op, Bin, Rel, Open, Close, Punct, Inner, Over, Under, Acc, Rad 和 Vcent。每个原子包含三个"区域", 即它的核, 上标和下标; 每个区域是空的, 或者由一个数学符号, 一个盒子或一个辅助数学列组成。数学符号依次有两个要素: 一个族编号和一个位置代码。

为了说清楚数学列, 要再引入几个语法规则:

⟨character⟩ ⟶ ⟨letter⟩ | ⟨otherchar⟩ | `\char`⟨8-bit number⟩ | ⟨chardef token⟩
⟨math character⟩ ⟶ `\mathchar`⟨15-bit number⟩ | ⟨mathchardef token⟩
  | `\delimiter`⟨27-bit number⟩
⟨math symbol⟩ ⟶ ⟨character⟩ | ⟨math character⟩
⟨math field⟩ ⟶ ⟨math symbol⟩ | ⟨filler⟩`{`⟨math mode material⟩`}`
⟨delim⟩ ⟶ ⟨filler⟩`\delimiter`⟨27-bit number⟩
  | ⟨filler⟩⟨letter⟩ | ⟨filler⟩⟨otherchar⟩

我们已经知道第二十五章中 ⟨character⟩ 的定义。的确, 字符是 TEX 的主要内容: 在水平模式下 TEX 消化过程所处理的所有命令的绝大部分是 ⟨character⟩ 命令, 它给出了 0 到 255 之间的一个数字, 告诉 TEX 把当前字体中相应的字符进行排版。当 TEX 处在数学模式或列表数学模式中时, ⟨character⟩ 命令还有更多的意思: 它给出了 0 到 $32767 = 2^{15} - 1$ 之间的一个数字。这是用字符的 `\mathcode` 值代替字符代码而完成的。但是如果 `\mathcode` 值是 32768 = ˝8000, 那么此 ⟨character⟩ 就被原来的字符代码(0 到 255)的一个活动符来代替; TEX 将忘掉原来的 ⟨character⟩, 并且按照第二十章的规则来展开此活动符。

一个 ⟨math character⟩ 通过直接用 `\mathchar` 或者以前的 `\mathchardef` 对应到一个 15 位数字上, 或者对应到一个 27 位数字的 `\delimiter`上; 在后一种情况下, 最不要紧的 12 位数字要被扔掉。

接着, 按照上面语法的定义, 每个 ⟨math symbol⟩ 给出了一个 15 位数字, 即, 0 到 32767 之间的一个数字。这样的数字可以用 $4096c + 256f + a$ 的格式了表示, 其中 $0 \le c < 8$, $0 \le f < 16$ 以及 $0 \le a < 256$。如果 $c = 7$, 那么 TEX 把 c 变成 0; 并且此时如果 `\fam` 的当前值在 0 到 15 之间, 那么 TEX 还要用 `\fam` 来代替 f。在所有情况下, 这个过程得到的类编号 c 都在 0 到 6, 族编号 f 在 0 到 15, 位置编号 a 在 0 到 255。(在 `\everymath` 和 `\everydisplay` 的刚开头, TEX 都暗中给出指定'`\fam=-1`'以给出 `\fam` 的初始值。这样, 只有用户在公式中明确改变 `\fam` 时, 才出现 f 的 `\fam` 改变。)

⟨math field⟩ 是用来给出原子的核, 上标和下标的。当 ⟨math field⟩ 是一个 ⟨math symbol⟩ 时, 此符号的 f 和 a 的编号就放在原子区域中。否则, 此 ⟨math field⟩ 以‘`{`’开始, 这使得 TEX 进入一个新层次的编组并开始一个新的数学列; 接下来的 ⟨math mode material⟩ 以‘`}`’来结束, 此时组也结束, 并且所得到的数学列放在这个原子区域中。如果发现此数学列是一个简单的单个 Ord 原子而没有上下标, 后者是核为 Ord 的一个 Acc, 那么要把封装所用的大括号去掉。

⟨delim⟩ 是用来定义 f 族中的“小字符” a 和 g 族中的“大字符” b 的, 其中 $0 \le a,b \le 255$ 并且 $0 \le f,g \le 15$; 这些字符代码是用来构造可变大小的分界符的, 见附录 G 的讨论。如果此 ⟨delim⟩ 由 27 位数字明确给出, 那么要得到的代码, 就要把这个数字拆解为 $c \cdot 2^{24} + f \cdot 2^{20} + a \cdot 2^{12} + g \cdot 2^8 + b$, 并且忽略掉 c 的值。否则, 此分界符由一个 ⟨letter⟩ 或 ⟨otherchar⟩ 记号给出, 并且此字符的 24 位 `\delcode` 值解释为 $f \cdot 2^{20} + a \cdot 2^{12} + g \cdot 2^8 + b$。

现在我们来讨论 TEX 在数学模式下使用的各个命令, 首先讨论的是与垂直和/或水平模式可类比的命令。

- `\hskip`⟨glue⟩, `\hfil`, `\hfill`, `\hss`, `\hfilneg`, `\mskip`⟨muglue⟩. 把一个粘连项目追加到当前数学列。

- ⟨leaders⟩⟨box or rule⟩⟨mathematical skip⟩. 这里的 ⟨mathematical skip⟩ 指的是刚刚提到的六个追加粘连的命令; ⟨leaders⟩ 和 ⟨box or rule⟩ 的正式语法见第二十四章。由引导符生成的粘连项目要追加到当前列。

- `\nonscript`. 一个宽度为零的特殊粘连项目被追加到当前列; 如果它后面的项目是粘连或者如果 `\nonscript` 最终被排在“标号字体”或“小标号字体”中,那么它就抵消了此列中的这个项目。

- `\noboundary`. 这个命令是多余的, 因此没什么用; 在数学模式中, 边界带子是自动无效的。

- ⟨space token⟩. 在数学模式中空格没有输出结果。

- `\␣`. 命令控制空格把粘连追加到当前列, 其大小与间距因子为 1000 时, 在水平模式下 ⟨space token⟩ 所插入的量是一样的。

- ⟨box⟩. 构造这个盒子, 当所得到的盒子为空时什么也不生成。否则, 把新的 Ord 原子追加到当前数学列, 并且此盒子变成它的核。

- `\raise`⟨dimen⟩⟨box⟩, `\lower`⟨dimen⟩⟨box⟩. 它与一个普通的 ⟨box⟩ 命令是一样的, 但是放这个核的新盒子要向上或向下移动给定的量。

- `\vcenter` ⟨box specification⟩ `{`⟨vertical mode material⟩`}`. 就象‘`\vcenter`’是‘`\vbox`’一样生成一个 vbox。接着, 把一个新的 Vcent 原子追加到当前数学列, 并且这个盒子变成它的核。

- ⟨vertical rule⟩. 把一个标尺追加到当前列(不看作一个原子)。

- `\halign`⟨box specification⟩`{`⟨alignment material⟩`}`. 这个命令只允许出现在列表数学模式中, 并且只有在当前数学列为空时才可以。这个对齐看作与封装垂直模式中那样来处理(见第二十四章), 只是这些行要向右缩进 `\displayindent`。闭符号‘`}`’后面可能要跟可选的 ⟨assignment⟩ 命令, 其后必须以‘`$$`’结束此列表公式。在所得到的对齐前后, TEX 要插入粘连 `\abovedisplayskip` 和 `\belowdisplayskip`。

- `\indent`. 宽度为 `\parindent` 的空盒子作为一个新 Ord 原子的核而被追加到当前列。

- `\noindent`. 这个命令在数学模式下没用。

- `{`⟨math mode material⟩`}`. 一个类代码为 1 的字符记号, 或者被 `\let` 为这样的字符记号的控制系列, 比如象 `\bgroup`, 它使得 TeX 开始一个新层次的编组, 并且还要开始建立一个新的数学列。当这样的组以'`}`'结束时, TeX 把所得到的数学列当作一个新的 Ord 原子的核, 并且此原子被追加到当前列。如果所得到的数学列是一单个 Acc 原子, 无论如何(即, 不管重音的长度如何), 要把此原子自己追加到当前列。

- ⟨math symbol⟩. (这是数学模式中最普遍的命令; 见本章邻近开头的语法。) 如前讨论, 每个数学符号给出三个值, c, f 和 a。 TeX 把一个原子追加到当前列, 其中原子的类型为 Ord, Op, Bin, Rel, Open, Close 或 Punct, 相应的 c 的值为 0, 1, 2, 3, 4, 5 或 6。此原子的核是由 f 和 a 定义的数学符号。

- ⟨math atom⟩⟨math field⟩. 命令 ⟨math atom⟩ 是下列任一个:

  `\mathord` | `\mathop` | `\mathbin` | `\mathrel` | `\mathopen`
  | `\mathclose` | `\mathpunct` | `\mathinner` | `\underline` | `\overline`

TeX 先处理此 ⟨math field⟩, 接着把所确定类型的一个新原子追加到当前列; 此原子的核包含了所确定的域。

- `\mathaccent`⟨15-bit number⟩⟨math field⟩. 当 TeX 处理任何 `\mathchar` 时, 都把 ⟨15-bit number⟩ 转换为 c, f 和 a。接着再处理 ⟨math field⟩ 并且把一个新的 Acc 原子追加到当前列。此原子的核包含了所确定的域; 在此原子中加重音的字符有自己的 (a, f)。

- `\radical`⟨27-bit number⟩⟨math field⟩. 当 TeX 处理任何 `\delimiter` 时, 都把这个 ⟨27-bit number⟩ 转换为 a, f, b 和 g。接着它处理这个 ⟨math field⟩ 并且把一个新的 Rad 原子追加到当前列。此原子的核包含了所确定的域; 此原子中的分界符这个域有自己的 (a, f) 值和 (b, g) 值。

- ⟨superscript⟩⟨math field⟩. ⟨superscript⟩ 命令是类代码为 7 的显式或隐式字符记号。如果当前列不是用一个原子结束, 那么将把所有域都为空的一个新的 Ord 原子追加上; 因此在所有情况下, 当前列都是以一个原子结束。此原子的上标域应该是空的; 当把它变成所给定 ⟨math field⟩ 的结果后, 它就是非空的。

- ⟨subscript⟩⟨math field⟩. ⟨subscript⟩ 命令是类代码为 8 的显式或隐式字符记号。它与 ⟨superscript⟩ 命令基本上是一样的, 当然其中要用下标域来代替上标域。

- `\displaylimits`, `\limits`, `\nolimits`. 只有当当前列以一个 Op 原子结尾时才能用这些命令。它们修改此 Op 原子中的一个特殊域, 以确定上下限采取什么样的约定。此域的正常值是 `\displaylimits`。

- `\/`. 把一个宽度为零的 kern 追加到当前列。(如果前一个字符的 italic 修正没有被正确添加, 它就给补加上。)

- `\discretionary`⟨general text⟩⟨general text⟩⟨general text⟩. 这个命令与水平模式中的处理方式一样(见第二十五章), 但是第三个 ⟨general text⟩ 必须得到一个空列。

- `\-`. 这个命令一般等价于'`\discretionary{-}{}{}`'; 因此这个'`-`'要当作连字符而不是减号。 (见附录 H。)

■ `\mathchoice`⟨general text⟩⟨general text⟩⟨general text⟩⟨general text⟩. 这四个通用文本的每个都被看作子公式(即, 象 ⟨math field⟩ 的定义中的第二个选项)。用这种方法定义的四个数学列被记录在一个"选择项目"中, 把此项目追加到当前列。

■ `\displaystyle`, `\textstyle`, `\scriptstyle`, `\scriptscriptstyle`. 与所确定字体相对应的项目被追加到当前列。

■ `\left`⟨delim⟩⟨math mode material⟩`\right`⟨delim⟩. TEX 开始一个新的组, 并且把 ⟨math mode material⟩ 放在以第一个分界符为左边界的一个新数学列中。这个组以 '`\right`' 结束, 此时内部数学列以第二个分界符为右边界而全部完成。接着 TEX 把一个 Inner 原子追加到当前列; 此原子的核包含了这个内部数学列。

■ ⟨generalized fraction command⟩. 这个命令来自下面六个形式中的一个:

`\over` | `\atop` | `\above`⟨dimen⟩
　　| `\overwithdelims`⟨delim⟩⟨delim⟩
　　| `\atopwithdelims`⟨delim⟩⟨delim⟩
　　| `\abovewithdelims`⟨delim⟩⟨delim⟩⟨dimen⟩

(见第十七章。) 当 TEX 遇见 ⟨generalized fraction command⟩ 命令时, 它把整个当前列取出并且放在一个广义分数项目的分子域中。这个新项目的分母域暂时为空; 左右分界符域设置为所给定的分界符的代码。TEX 把这个广义分数项目保存在与当前所处理的数学模式的层次相应的一个特殊地方。(在此特殊地方, 不应该有其它广义分数项目, 因为象 '`a\over b\over c`' 这样的命令是非法的。) 接着 TEX 把当前列变成空的, 并且继续在数学模式下处理指令。稍后, 当数学模式的当前层次完成后(出现一个'`$`', '`}`'或者 `\right`, 这与当前组的性质有关), 当前列将被移到所保存的广义分数项目的分母域中; 接着这个项目把自己整个变成一个完整的列。但是, 在当前列以 `\left` 开始且以 `\right` 结束的特殊情形下, 边界项目将会从广义分数的分子和分母中提取处理, 并且最后的列由三个项目组成: 左边界, 广义分数, 右边界。(如果你要查看数学列构建的过程, 可以在 TEX 处理广义分数的分母时声明'`\showlists`'。)

■ ⟨eqno⟩⟨math mode material⟩`$`. 这里的 ⟨eqno⟩ 表示 `\eqno` 或者 `\leqno`; 这些命令只允许出现在列表数学模式中。碰到 ⟨eqno⟩ 后, TEX 进入一新层次的编组, 插入记号 `\everymath`, 并且进入非列表数学模式以把 ⟨math mode material⟩ 放到数学列中。当此数学列结束时, TEX 把它转换为水平列, 并且把结果放在一个盒子中, 此盒子就是当前列表公式的方程编号。闭记号 `$` 将被放回到输入中中止列表公式的地方。

■ `$`. 如果 TEX 处在列表数学模式, 它还要再读入一个记号, 并且必须还是 `$`。在任一情况下, 这个数学转换命令都终止正在处理的数学模式的当前层次, 并且结束当前组(此组应该是以 `$` 或者 ⟨eqno⟩ 所开始的)。一旦数学列结束了, 就被转换到水平列中, 就象附录 G 中讨论的那样。

■ 除上面的以外: 如果任何其它的 TEX 原始命令出现在数学模式中, 就给出错误信息, 并且 TEX 试着用一种合理的方法来修复它。例如, 如果出现了命令 `\par`, 或者是其它非数学的命令, 那么 TEX 就试着在这样的记号紧前面插入一个'`$`'; 这就退出了数学模式。在另一方面, 如果象 `\endcsname`, `\omit` 或 `#` 这些完全不能出现在数学模式中的记号, TEX 就在给出错误信息后直接忽略掉它。可能你喜欢输入一

些的确错误的命令来看看会出现上面情况。(为了得到最全的信息, 象第二十七章讨论的那样, 首先要声明‘\tracingall’。)

▶ **练习26.1**
10 的幂: 整个 TEX 语言已经汇总完毕。为了看看你掌握了多少, 请你能指出数字 10, 100, 1000, 10000 和 100000 对 TEX 有特殊的意思所有地方。

▶ **练习26.2**
2 的幂: 请你指出数字 8, 16, 32, 64, 128, 256, . . . 对 TEX 有特殊意思的所有地方。

27. 错误修复

好了, 所有你应该知道的 TEX 的东西都讨论完了——只要你不爱出错。如果你预计不出错, 那么不用费心来看本章。否则, 看看 TEX 怎样定位你的文稿中的破绽会很有帮助。

在第六章中的实战中, 你已经知道了错误信息的一般形式, 并且学会了对付 TEX 找到的毛病的几种方法。在实践中, 你可以"实时"修正大多数错误, 只要 TEX 检测到它们, 就可以插入或删除一些东西。对待它的正确方法是在你处理 TEX 时要老练, 并且把错误信息看作有趣的难题——"为什么计算机会这样?"——而不是觉得有失体面。

TEX 能给出一百多种的错误信息, 可能你从不会遇见所有的这些, 以外有些错误很难出现。在第六章我们讨论了错误"undefined control sequence"; 现在我们再看看其它几个。

如果你拼错了某些测量单位——例如, 如果你输入的是'`\hsize=4im`'而不是'`\hsize=4in`'——就得到象下面这样的错误信息:

```
! Illegal unit of measure (pt inserted).
<to be read again>
                   i
<to be read again>
                   m
<*> \hsize=4im
               \input story
?
```

在 TEX 处理之前要读入应该合法的单位; 因此在这种情况下它就在输入的当前位置处暗中插入了'`pt`', 并且设置 `\hsize=4pt`。

修复这种错误的最好方法是什么呢? 嗯, 如果你不明白错误信息的意思, 总可以键入'`H`'或'`h`'来查看帮助信息。接着你可以查看前后的行, 并且看看是否你直接键入 ⟨return⟩ TEX 就依次读入'`i`''`m`'以及'`\input story`'并且进行下去。不幸的是, 这种简单的方法并不很好, 因为'`i`' 和 '`m`'将作为新段落的一部分进行排版。这种情况下最佳的修复方案应该是首先键入'`2`'。这就告诉 TEX 把接下来要读入的两个字符扔掉; 并且在 TEX 这样做完后, 它就再次停下来让你查看新的状态。你将看到的如下:

```
<recently read> m

<*> \hsize=4im
               \input story
?
```

好; '`i`'和'`m`'已经被读入并且扔掉了。但是如果你现在键入 ⟨return⟩, TEX 就将'`\input story`'并且按照 `\hsize=4pt` 来排版 `story.tex`; 那就没什么特别之处了, 因为它直接得到了几打溢出的盒子, 每个都是 story 的一个音节。还有更好的方法: 你可以插入你所要的命令, 即现在键入

```
I\hsize=4in
```

这就告诉 TeX 把 `\hsize` 改为正确的值, 其后将 `\input story`, 并且符合你的要求。

▶ **练习27.1**

用户笨笨在回应刚才出现的错误信息时键入了‘8’而不是‘2’; 他要删除‘i’, ‘m’, ‘\input’, 以及‘story’的五个字符。但是 TeX 的回应为

```
<*> \hsize=4im \input stor
                          y
```

看看出现了什么问题。

TeX 通常修复错误的方式是忽略掉它不认识的命令, 或者插入某些东西来去掉错误。例如, 在第六章我们看到 TeX 忽略掉一个没有定义的控制系列; 还有, 我们刚刚看到, 当 TeX 需要一个测量的物理单位时, 就插入‘pt’。在下面的例子中, TeX 将插入某些东西:

```
! Missing $ inserted.
<inserted text>
               $
<to be read again>
                   ^
l.11 the fact that 32768=2^
                           {15} wasn't interesting
? H
I've inserted a begin-math/end-math symbol since I think
you left one out. Proceed, with fingers crossed.
```

(用户落掉了封装公式的符号 $, TeX 就通过插入它来修复。) 在这种情况下, ⟨inserted text⟩ 明确给出了, 并且还没有读入; 相反, 在前一个例子中 TeX 已经内在地插入了‘pt’。因此, 在 TeX 实际读入‘$’前用户可以有机会去掉这个插入的‘$’。

应该怎样做呢? 本例的错误出现在 TeX 发现错误信息之前; 字符串‘32768=2’已经在水平模式下排版完毕。无法回去取消过去的结果, 因此在‘=’两边落掉的间距就无法弥补了。因此现在我们修复错误的目的不是得到完美的输出结果; 我们更希望用某种方式继续进行, 而 TeX 将越过现在的错误并且去检测下一个错误。如果我们现在直接键入 ⟨return⟩, 目的就会落空, 因为 TeX 将把随后的文本按照数学公式排版: ‘$^{15}wasn'tinteresting\ldots$’; 当在闭符号‘$’出现前段落就结束时就又出现一个错误。另一方面, 有一种更精巧的修复方法, 即键入‘6’并且接着键入‘`I$^{15}$`’; 这就删除了‘`$^{15}`’并且插入一个正确的公式部分。但是它比必需的要更复杂。这种情况下最好的解决方法是只键入‘2’, 并且接下来继续则可; TeX 将排版出一个不正确的方程‘32768=215’, 但是重要的是你能验证一下文档其余的部分, 就好像没出现过这个错误一样。

练习27.2

下面这种情况下, 无意中落掉一个反斜线:

```
! Missing control sequence inserted.
<inserted text>
                \inaccessible
<to be read again>
                   m
l.10 \def m
           acro{replacement}
```

在'\def'后面 TEX 要读入一个控制系列, 因此它插入了一个可以继续进行处理的命令。(这个控制系列显示的是'\inaccessible', 但是它与你的无错的文稿中实际给出的任何控制系列都没有关系。) 如果此时你直接键入 ⟨return⟩, 那么 TEX 将定义一个不能使用的控制系列, 但是这没什么好处; 随后要用到的 \macro 就没有定义了。看看怎样修复这个错误才能使得第 10 行的结果为'\def\macro{replacement}'

练习27.3

当你用选项'I'回应错误信息时, 第八章的规则意味着 TEX 将从行的右端去掉所有空格。看看怎样才能绕过这个规则而用选项'I'插入一个空格?

要对付的某些最难的错误是这样的, 错误出现在(比如说)第 20 行, 但是 TEX 直到第 25 行左右才告诉你出了问题。例如, 如果你落掉了'}', 而它表示某些宏的变量的结束, 那么直到下一个段落的结束 TEX 才会发现这个问题。在这种情况下, 你可能已经无法挽回这个段落了; 但是 TEX 一般可以及时纠正来处理随后的段落, 就象什么也没有发生一样。将会出现一个"失控变量", 并且通过查看此文本的开头可以找到所落掉的'}'在哪里。

聪明点就知道, 文档中的第一个错误可能会造成后面大量虚假的"错误", 因为有毛病的命令会使得 TEX 处理随后命令的能力遭到极大损害。但是大多数情况下你会发现依次运行就可以找到你的输入与 TEX 规则冲突的所有地方。

当错误来自理解偏差而不是笔误时, 问题就更严重了: TEX 的帮助信息可能没有太多的用处, 即使能给出一些。如果你无意中重新定义了一个重要的控制系列——例如, 如果你声明'\def\box{...}'——那么可能出现各种奇怪的错误。计算机没这个能力, 并且 TEX 只按照它自己的方法来处理; 如果你不了解机器的逻辑, 那么这样的解释就觉得难以理解。当然, 解决之道是请教别人; 或者使用杀手锏——看看第 2, 3, . . ., 26 章中的指令。

练习27.4

J. H. Quick (一个学生)曾经定义了下列一组宏:

```
\newcount\serialnumber
\def\firstnumber{\serialnumber=0 }
\def\nextnumber{\advance \serialnumber by 1
  \number\serialnumber)\nobreak\hskip.2em }
```

这样它就可以这样输入——例如

```
\firstnumber
\nextnumber xx, \nextnumber yy, and \nextnumber zz
```

并且 TEX 应该排版出下列结果: '1) xx, 2) yy, and 3) zz'。好, 这不错, 他把这些宏教给几个同伴。但是几个月后, 他接到一个紧急电话; 他的一个朋友遇到一个非常奇怪的错误信息:

```
! Missing number, treated as zero.
<to be read again>
                  c
l.107 \nextnumber minusc
                        ule chances of error
?
```

看看出现了什么问题, 并且给出 Quick 一个建议。

早晚——希望是早——你用 TEX 处理整个文件会顺利完成。但是可能输出结果还不对; TEX 不停顿并不意味着你可以不用校对。此时, 更容易通过改正输入来去掉笔误。版面错误可以通过前面讨论的方法来对付: 溢出的盒子可以用第六章的方法来解决; 断点不好可以通过带子命令或 `\hbox` 命令来避免, 就象第十四章讨论的那样; 数学公式可以用第十六至十九章的原则来改进。

但是输出结果可能包含一些看起来比较费解的错误。例如, 如果你使用了打印设备不支持的某些放大率的字体, 那么 TEX 不知道有什么问题, 但是把 dvi 文件转换为输出结果的程序可能没有告诉你它用一种"近似"字体代替了你实际需要的字体; 这样就出现了非常难看的多余间距。

如果无法找到错误的地方, 就试着简化你的程序: 去掉所有正常的部分, 直到得到出现同样问题的最短输入文件。文件越短, 越容易找到错误。

可能你感到纳闷, 为什么 TEX 在你要输入空格的某些地方没有放置空格。记住, 在读入文件时, TEX 要忽略掉控制单词后面的空格。(TEX 还忽略掉作为原始命令的变量而出现的 ⟨number⟩ 或 ⟨unit of measure⟩ 后面的空格; 但是如果你使用的是设计正确的宏, 那么这些规则与你无关, 因为你可能不会直接用到原始命令。)

另一方面, 如果你在设计宏, 那么处理问题的任务就更复杂了。例如, 你可能发现, 当 TEX 处理由几打命令组成的某些复杂的长系列代码时, 可能出现了三个空格。怎样你才能找到这些空格的来源呢? 这就要用第十三章中提到的'`\tracingcommands=1`'。这使得 TEX 只要执行一个原始命令, 就在 log 文件中放一条记录; 这样你就能看到什么时候命令是'`blank space`'了。

大多数 TEX 的执行都允许用某些方式来中断程序。这就使它可以诊断出死循环的根源。当中断时, TEX 转到 `\errorstopmode`; 因此, 你就有机会把命令插入到输入中: 可以结束运行, 也可以 `\show` 或改变控制系列, 寄存器等的当前定义。如果你凑巧使用了一个效率低下的宏, 你还可以觉察出 TEX 把大多数时间都花在了什么地方了, 因为随机出现的中断倾向于出现在 TEX 最频繁读入的地方。

有时候错误太糟糕使得 TEX 被迫过早退出。例如, 如果在 \batchmode 或 \nonstopmode 运行, 那么如果 TEX 需要从终端得到输入时就“紧急中止”了; 当必须打开的文件被打开时, 或者在输入文档中找不到命令 \end 时, 就出现这种情况。下面就是 TEX 在中止运行时你可能得到的信息:

```
Fatal format file error; I'm stymied.
```

这表示你要预载入的格式不能使用, 因为它是为 TEX 的另一个版本所制作的。

```
That makes 100 errors; please try again.
```

自上一段结束起, TEX 已经发现了 100 个错误, 因此它可能是个死循环。

```
Interwoven alignment preambles are not allowed.
```

如果你错到这种情况, 就应该去学习了, 不值得同情。

```
I can't go on meeting you like this.
```

前面的错误使得 TEX 无法正常运行。改正它并且再试运行一次。

```
This can't happen.
```

正在使用的对 TEX 而言是错误的。问题太大。

还有一个可怕的信息 TEX 极不情愿给出。但是也会出现:

```
TeX capacity exceeded, sorry.
```

唉, 这意味着你所展开的 TEX 太长。这个信息将告诉你 TEX 的哪部分内存溢出了; 下面十四种情况之一会出现:

`number of strings` (文件或控制系列的名称)
`pool size` (在这样的名称中的字符)
`main memory size` (盒子, 粘连, 断点, 记号列, 字符等。)
`hash size` (控制系列的命名)
`font memory` (字体度量数据)
`exception dictionary` (额外的连字)
`input stack size` (同时输入的资源)
`semantic nest size` (要构造的未完成的列)
`parameter stack size` (宏参数)
`buffer size` (在要读入文件中行中的字符)
`save size` (在组结束时要复原的值)
`text input levels` (`\input` 文件和错误的插入对象)
`grouping levels` (未完成的组)
`pattern memory` (连字式样的数据)

当前可用的内存量也显示出来。

如果有一个未溢出 TEX 容量的任务, 但是你只想看看离限制有多远, 那么就在任务结束前把 `\tracingstats` 设置为一个正值。这样在 log 文件中就对上述给出的前十一项(即, number of strings, . . ., save size)按相应顺序给出实际使用的一个报告。还有, 如果把 `\tracingstats` 设置为大于等于 2, 那么 TEX 只要执行一次命令 `\shipout` 就显示一次当前内存。这样的统计包括分为两部分; 例如, '490&5950'表示对象盒子, 粘连和断点这样的"大"对象用到 490 个单词, 而对象记号和字符这样的"小"对象用到 5950 个单词。

如果超出了 TEX 的容量怎么办? 如果你的计算机容量足够大, 那么上面列出的所有容量分量都可用增加; 实际上, 在不增加 TEX 的总容量情况下, 要给一个分量增加所必需的空间一般要减小其它某些分量。如果你特别重要的用处, 可以请求你的局域系统给你一个可手工设置容量的特殊 TEX。但是在进行这样的步骤前, 首先你必须能正确使用 TEX。如果你要排版超过一页的一个巨大段落或者对齐格式, 那么可能要改变你的方法, 因为在 TEX 完成断行或对齐计算前要把它们都读入; 这要消耗大量的内存容量。如果你已经建立了一个大的宏库, 那么就应该记住, TEX 必须记住你所定义的所有的替换文本; 因此如果内存空间较小, 就只载入所用的宏。(至于怎样使宏更紧凑, 见附录 B 和 D。)

有些错误的 TEX 程序会溢出所有的有限内存容量。例如, 在'`\def\recurse{(\recurse)}`'后, 使用 `\recurse` 就立即溢出:

```
! TeX capacity exceeded, sorry [input stack size=80].
\recurse ->(\recurse
                    )
\recurse ->(\recurse
                    )
...
```

不管你怎样增加 TEX 的输入堆栈的容量, 总会明显出现同样的错误。

"`save size`"容量溢出的特殊情况是一个最难对付的错误, 特别是你只在大任务中出现这种错误。只要 TEX 所执行的某些量的非全局指定, 而这些量的前一个值不是在同一层次的编组中指定的, 那么就要保存为两个字。当宏编写正确时, 在"保存堆栈"上极少情况下才能用到 100 个以上的位置; 但是如果把局域和全局变量都指定给同一变量, 可能导致保存所用的堆栈无限制地增长。通过设置 `\tracingrestores=1` 可以查看 TEX 在保存堆栈上放了什么; 这样 log 文件就记录下了在组结束时从堆栈中所移除内容的信息。例如, 设 `\a` 表示命令'`\advance\day by 1`'; `\g` 表示'`\global\advance\day by 1`'; 看看下列命令:

```
\day=1 {\a\g\a\g\a}
```

第一个 `\a` 设置 `\day=2` 并且把原来的值 `\day=1` 放在保存堆栈中从而保留它。第一个 `\g` 全局地设置 `\day=3`; 在全局指定时, 堆栈上不需要什么变动。下一个 `\a` 设置 `\day=4` 并且把原来的值 `\day=3` 放在保存堆栈中以保留。接着, `\g` 设置 `\day=5`; 接着 `\a` 设置 `\day=6` 并且保留住 `\day=5`。最后, '}'使得 TEX 回退

这整个保存堆栈; 如果在此处声明 `\tracingrestores=1`, 那么 log 文件中就得到下列数据:

```
{restoring \day=5}
{retaining \day=5}
{retaining \day=5}
```

注解: 参数 `\day` 首先恢复其全局值 5。因为此值是全局的, 所以要保留下来, 因此其它两个被保存的值 (`\day=3` 和 `\day=1`) 就被忽略了。教训: 如果你发现 TEX 保留了很多值, 那么就有那么一组宏, 在大文件中会导致保存堆栈溢出。为了防止这种情况, 最好保持对每个变量的指定要前后一致; 指定应该总是全局的, 或者总是局域的。

除了 `\tracingstats` 和 `\tracingrestores` 外, TEX 还有几种跟踪命令: 在第十三和二十章我们已经讨论过 `\tracingcommands`, 在第十四章中有 `\tracingparagraphs`, 在第十五章中有 `\tracingpages`, 在第二十章中有 `\tracingmacros`。还有 `\tracinglostchars`, 在其为正值时, 如果因为当前字体中不存在某个字符而漏掉了字符, TEX 就给出一个记录; 以及 `\tracingoutput`, 在其为正值时, TEX 就以符号的形式把要输送到 `dvi` 文件中的某个盒子的内容显示出来。在你无法确定问题出在 TEX 还是后处理软件方面时, 用后一个命令可以看看排版是否正确。

当 TEX 把一个盒子作为诊断输出的一部分而显示出来时, 可以用两个参数来控制信息量, 它们是 `\showboxbreadth` 和 `\showboxdepth`。Plain TEX 把第一个设置为 5, 显示的是每个层次的项目的最大数目; 第二个设置为 3, 显示的是深度最大的层次。例如, 当 `\showboxbreadth=1` 和 `\showboxdepth=1` 时, 所有内容如下的一个小盒子

```
\hbox(4.30554+1.94444)x21.0, glue set 0.5
.\hbox(4.30554+1.94444)x5.0
..\tenrm g
.\glue 5.0 plus 2.0
.\tenrm | (ligature ---)
```

将简化为

```
\hbox(4.30554+1.94444)x21.0, glue set 0.5
.\hbox(4.30554+1.94444)x5.0 []
.etc.
```

如果设置 `\showboxdepth=0` 就只得到最顶层的信息:

```
\hbox(4.30554+1.94444)x21.0, glue set 0.5 []
```

(注意, ‘`[]`’和‘`etc.`表明数据被截去了。)

如果一个非空 hbox 的粘连不能再收缩到所要求的尺寸, 并且其 `\hbadness` 小于 100 或者(在最大收缩后)其超出的宽度大于 `\hfuzz`, 它就被看作是“溢出”的。如果粘连是收缩并且其 badness 超过了 `\hbadness`, 它就是“太紧”的; 如果粘连是伸长并且其 badness 超过了 `\hbadness` 而不超过 100, 就是“松散”的; 如果粘连是伸长并且其 badness 超过了 `\hbadness` 且超过 100, 就是“空荡”的。类似讨论对非空 vbox

也适用。只要出现这样的问题，TEX 就给出警告信息并且显示出有毛病的盒子。 空盒子从不认为有这种问题。

当对齐对象是“溢出”，“太紧”，“松散”或“空荡”时，不会为每个要对齐的行给出警告信息，只能得到一个，并且 TEX 显示出一个模板行(在 \valign 中为一个模板栏)。例如，假定输入的是‘\tabskip=0pt plus10pt \halign to200pt{&#\hfil\cr...\cr}’，并且假定要对齐的内容把两个栏宽度分别变成 50 pt 和 60 pt。那么就得到下列信息:

```
Underfull \hbox (badness 2698) in alignment at lines 11--18
 [] []
\hbox(0.0+0.0)x200.0, glue set 3.0
.\glue(\tabskip) 0.0 plus 10.0
.\unsetbox(0.0+0.0)x50.0
.\glue(\tabskip) 0.0 plus 10.0
.\unsetbox(0.0+0.0)x60.0
.\glue(\tabskip) 0.0 plus 10.0
```

在模板行中的“unset box”显示的是各个栏的宽度。在本例中，为了伸长到所要求的 200 pt，制表粘连的伸长量为其伸长值的三倍，因此此盒子是空荡的。(按照第十四章的公式，此情形下的 badness 为 2700; TEX 实际用的是一个类似且更高效的公式，因此计算出的 badness 为 2698。) 对齐的每个行都是空荡的，但是只有模板行在警告信息中给出了。对溢出对齐的行不再添加“溢出的线”。

如果参数 \tracingonline 不是正值，那么 \tracing... 命令的所有输出都放在 log 文件中; 如果是正值，所有诊断信息都放在终端和 log 文件中。Plain TEX 有一个宏 \tracingall，它把各种诊断信息全部给出。它不但设置了 \tracingcommands，\tracingrestores，\tracingparagraphs 等，还设置了 \tracingonline=1，并且 \showboxbreadth 和 \showboxdepth 的值也非常大，这样就把所有盒子的全部内容都显示出来了。

某些 TEX 版本在速度上有所改进。这些运行看不到参数 \tracingparagraphs，\tracingpages，\tracingstats 和 \tracingrestores 的值，因为当 TEX 不必进行统计或者不理会诊断时运行得更快。如果要使用所有的诊断工具，就要用选对版本。

如果设置 \pausing=1，那么 TEX 将在读入文件时允许你编辑每一行。用这种方法，当你要处理问题时，就可以临时补救一下(比如插入命令 \show...)而不改变文件的实际内容，并且可以让 TEX 按照人为的速度来运行。

最后的提示: 当要制作一个大文稿时，最好每次只处理几页。设置一个临时文件和一个总文件，并且把文本放在分文件中。(把控制基本格式的控制信息放在此文件的开头; 在附录 E 中有一个例子 galley.tex。) 在临时文件正确后，再把它添加到总文件中; 这样你只需要偶尔编译总文件看看页面的合并情况。例如，当作者写此手册时，每次处理一章，并且把长的章分为几个片段。

▶ **练习27.5**

最后的练习: 找出本手册中所有的谎话和玩笑。

最 后 的 倡 议: 向 前, 创 造 出 版 业 之 杰 作 吧!